Introduction

The STORM Sound Kit was designed with several goals in mind. The Sound Manager that is included with Macintosh System Software is both too complicated and too limited for use in games. Most programmers will stumble and fall when faced with learning to use the Sound Manager. Furthermore, if you need to use sampled sound, the Sound Manager limits you to a single sound at a time.

The STORM Sound Kit allows two simultaneous sampled sounds. You only need to use four simple routines to play these sounds. The current implementation of the kit uses the old Sound Driver, but there will be a more compatible version that uses the Sound Manager routines. As it is now, the sound kit has almost no impact on CPU performance, although it is regularly called from an interrupt.

Installation into New Projects

First, you need to copy all resources of type ‘SKIT’ to your resource file. There should be three of them: 1000, 1001 and 1002. SKIT-1000 should have the system heap and locked attributes set. An error in the installation will cause strange and wonderful system crashes.

You then need to copy three source files (or a project made out of these) to your project file. The files are: SoundKit.c, Huffman.c and Datafiles.c. Make sure that the files Shuddup.h and Huffman.h are accessible as include files.

You now need to create a new version of the “-> Compression” application with ResEdit. The file name (or the path) of your resource file is stored in the STR 128 resource. Keep all your sound files and the compression application in the same folder. If you have a lot of sounds, you may have to increase the SIZE allocation for the compression routine. Allocate approximately two times the total size of the sounds.

Creating Sounds

Sounds can be created with industry standard sampled sound editors like SoundCap, SoundWave and SoundEdit (or even the SID utility). The Sound Kit limits playback to 11kHz, so that should be taken into account when creating new effects.

A special utility program (-> Compression) is used to compress a number of sound files into a resource format that can be used by the Sound Kit. This application doesn’t have a user interface and you have to know how to use it. Fortunately it is not an end user product and will probably be used only by programmers or experienced users who are well aquainted with resources. Compression source code is included with the distribution.

To create a new set of sounds, place all the sounds in the same folder with the application. Put your resource file in that same folder. Note that the name of the resource file is significant and that it is stored in a resource. If you rename your resource file, you need to change the compression application accordingly.

When all the files are there, launch -> Compression. It will briefly display a distribution graph that it uses to create the Huffman compression code. When the program returns, two resources in your resource file will have been uptdated to contain all the sounds in the folder.

Compression reduces the size of the sound data by about 40%. It does this by first dropping the least significant bit and then packing with a Huffman algorithm aided by delta-coding. (For more information, see the documentation for DeltaSound, available with anonymous ftp from vega.hut.fi and sumex-aim.stanford.edu.)

Using the Sound Kit

The sound kit is fairly easy to use, once you have it set up correctly. Sounds are accessed by a sound index that is determined by the name of the originaly sound file. The first sound in alphabetical order is sound 0, the second is 1, etc. To make it easier for you to actually use the sounds and add new sounds, it is recommended that you create an enum type for the sounds. Take a look at STORM.h for an example on how to do it.

You have to initialize the Sound Kit before using it. To do it, call SKInit. To play sounds on channel A, call PlayA and to play sound on channel B, call PlayB. When you are finished with the sound kit, you call call CloseSoundKit.

Here’s a short sample application:

```
#include "STORM.H"
```

```
void        main()
{
            DoInits();
            InitSoundKit();

                        PlayA(Swish,0);
                        PlayB(Bonk,0);
                        while(!Button());

                        PlayA(FlyThru,0);
                        while(Button());

            CloseSoundKit();
}
```

void InitSoundKit();

InitSoundKit loads the resources, decodes the Huffman packing, starts playing a sound and sets up a vertical blanking task to handle continuous sound mixing. Your application should make sure that there is enough memory available to make all these steps. Decoding the packing requires memory for both the compressed and uncompressed data!

InitSoundKit patches ExitToShell so that CloseSoundKit is called even if the program itself forgets to make the call.

It is recommended that you compile a list of sound files into an enum type so that the sounds can be accessed by name. Using numbers to identify sounds should be considered bad programming style.

There are two ways in which the sound kit can work. If OldSound (a global variable) is nonzero when InitSoundKit is called, the old Sound Driver is used and a vertical blanking task is installed to feed data to the driver.

The vertical blanking task is installed in the system heap. In order to be able to remove this task when the program quits (even if the program exits without calling CloseSoundKit), a system trap patch is installed so that CloseSoundKit is called in every possible case. Even though you do not have to, it should be considered good programming style to call CloseSoundKit anyway.

If OldSound is zero, the new Sound Manager is used. System 6.07 or later is absolutely required. The sound kit uses Gestalt to find out if the new sound manager is available and reverts to the sound driver, if it can’t use the sound manager. The Sound Manager uses more CPU, but doesn’t require a vertical blanking interrupt task.

OldSound is initially zero and unless you really want to use the old sound routines (to be compatible with old systems, for instance), you shouldn’t have to change it.

void CloseSoundKit();

CloseSoundKit silences the sound, removes the vertical blanking task and disposes of the sounds. CloseSoundKit removes the patch on ExitToShell, so if you have done any patches to ExitToShell while using the Sound Kit, they will also be removed. Tough luck.

void PlayA(int num,int priority);
void PlayB(int num,int priority);

PlayA and PlayB are similar routines that play sounds on channel A or B. The num parameter is a sound index to tell which sound to play and the priority can be used to give sounds priorities. A sound with a higher priority will interrupt a sound with a lower priority on the same channel. A sound with a lower priority will be ignored.

A special rule was added so that both channels are used more often. If a sound can not be played on a certain channel because a sound with a higher priority is playing, the other channel will be checked and if no sound is active on that channel, the new sound will be played on that channel with priority 0.

Note that you can play a sound at higher volume by using both sound channels simultaneously. There are no guarantees that the sounds will sync perfectly, but this is usually the case. Disable vertical interrupts first, if you wish to have perfect synchronization. (Remember to enable them immediately after you make the calls.)

Note that these routines are very simple and that you are free to implement a more sophisticated scheme of channel allocation. If you need more than two channels of sampled sound, please contact me. More channels should be easily possible, at the cost of dynamic range.

void SKVolume(int volume);

You can temporarily set the volume to something else than what the user set with the control panel by calling this routine. Volume 0 is silence and 8 is the highest possible volume. Sound volume is reset when the kit is closed.

void SKProcessHandle(Handle thedata);

Sometimes memory is so tight that you want to load sound dynamically. In order to play a sampled sound with the sound kit, the sound has to be preprocessed first. If you load your sound data into an **unlocked** handle, you can call SKProcessHandle to change it into something that can be played with the following two routines:

void SKPlayHandleA(Handle thesound, int priority);
void SKPlayHandleB(Handle thesound, int priority);

These routines are similar to PlayA and PlayB, but they take a preprocessed sound kit sound handle instead of a sound number.

Gotchas:

The sound kit doesn’t handle the packing and unpacking of your own sounds, so unless you write your own packing and unpacking code, you’ll waste some disk space with the unpacked sounds.

The sounds have to be sampled at 11 kHz.

The sound resource has to be locked before you call SKPlayHandle and it has to remain locked as long as the sound is playing. The only way to tell if a sound has completed playing is to compare **both** sound channel data pointers (stored in Vv.ChanA and Vv.ChanB) to the area that your resource is occupying. Unlocking the sound resource before it has completed playing will probably result in garbled sound.

Ideas:

It’s possible to change the sound kit to load multiple sound sets and unload them too. You just have to change to the correct set before calling PlayA and PlayB and you have to handle the unloading yourself. I’ll implement this in the future, if there really is need for trickery like this.

Appendix - Extract from an E-mail Message

[*This is an extract of a mail message that I sent to one of the testers are a response to his question why there are no ‘snd ‘ resources in the STORM application.*]

The simple reason why I'm not using 'snd '-resource is that STORM 0.9A3 doesn't use the sound manager. It uses the old drivers. The latest version (0.9A5) knows about the sound manager, but still allows the use of the old routines (they are faster, as I explained in another mail).

You might also have noticed that STORM produces two simultaneous channels of sampled sound. The way I'm doing this is similar for the sound driver and manager.

To mix two channels of sound, I just add the signals and play the result. Apple does this by adding the signals and then dividing by the number of channels that are playing (this is with the 6.07 sound manager). This means they have to continually divide values. My sound samples are divided by two in advance. This has two advantages: division (or shifting) is not needed *and* the additions can be done with four *parallel* additions at a time.

If you know that your source values are between 0 and 127, then you can add four values at a time with a single long addition. It also means that you read and write 4 bytes at a time, so memory access is also faster than reading a byte at a time.

Let's first talk about the old sound driver.

There used to be a real technical note number 19. Now there's replacement that basically says that you shouldn't use the old technical note 19 (which is the reason that I now support the Sound Manager...I just didn't have good documentation on the sound manager before last week, when I got a peek at Inside Mac VI).

Technote 19 says that you produce sound without a click by playing a single sound continuously and changing the active count of the parameter block on the fly. You can also modify the contents of the sound buffer.

STORM installs a 60 Hz VBL routine that flips the ioActCount field between two 185-byte buffers. 185 bytes make 1/60th of a second at 11kHz, which is the sample rate that is currently used.

Now you might have noticed that having a 185 byte buffer size means that only one buffer out of four is long-word aligned. The solution to this is to use 188 bytes instead of 185. The remaining 3 bytes are never played, so I just set them to something convenient (the last value is repeated). This is done when the sounds are loaded, so when I use the sound driver routines, I have the sounds segmented into 188 byte blocks. Memory reads are long-word aligned, writes are long-word aligned (the buffers are really 188 bytes. The sound driver never sees more than 185).

So, the timings are quite delicate, but things work fine with most computers (including the MacPlus & other classics). To play a new sound, I just modify two variables that the VBL task is using (being careful to first lock the VBL task from changing these variables).

So, how do things work with the sound manager?

The new and improved sound manager that is shipped with 6.07 and 7.0 allows the use of double buffers. I was very excited to see this implemented, since it was exactly what I was doing. The surprise was that double buffering buzzed badly, if the buffers were smaller than 1024 samples at 22kHz or 512 samples at 11kHz. I just changed my routines to use 512 byte buffers when the sound manager is used.

No VBL task is needed with the sound manager, so the task isn't installed. I haven't tried the new routines on the machine where the old ones didn't work, but there's still a chance that the new ones won't work either. (I think they will work, since with the sound manager buffering, you always have two 2048 samples at 22kHz available, so being late with interrupt handling shouldn't cause problems.)

If this were all, I could still use 'snd ' resources and everyone could happily steal the sounds from STORM.

Originally the sounds were stored in the funny 185 out of 188 format that is now used in RAM only. Since there are now 200 kB worth of sounds and other resources take up another 80 kB, why is the total size of the application only about 200 kB?

The answer is that wrote a Huffman (howmanyf'sandn'sIneverknow) compression routine that first divides by two, then calculates delta values and then compresses a number of files into two resources. One resource is a directory of sounds (just sound sizes in samples) and the other is a frequency table with the compressed data appended. This routine reduces the size of sound files by about 45%. Ok, in this case it's only 38%, but it's still good. Compact Pro only achieves 13% and my method is not quite as lossy as that one used with MACE. The compression code takes 2500 bytes.)

I just changed the compression routines so that the heap is less fragmented than it used to be. This shouldn't have any effect on the game, but it does make the decompression time slightly longer (I couldn't notice the difference).

---------- Footnotes ----------

1. The resource file also needs to be in this same folder, unless you specify a full path for theresource file in the STR resource.

---------- Sidebars ----------

Created: Saturday, October 27, 1990 Last change: Thursday, October 10, 1991

Project STORM

Author: Juri Munkki

STORM Sound Kit

This document describes the programming interface to a set of routines that allow sampled sound output with two channels. A sampling rate of 11 kHz is used and each sample has 7 bit resolution.