# Technote 1108

## Unknown Sound Features

**CONTENTS**

Many developers have complained about features they felt were lacking in the Sound Manager. However, many features which were believed to be lacking were actually available, just under documented. This Technote is meant to document the more obscure features of the Sound Manager.

This Technote is directed at application and hardware developers who work with the Sound Manager and want to be sure that they are getting the most out of the Sound Manager and their products.

## The little known features of the Sound Manager

The Sound Manager features that this Technote talks about are:
Multiple Sound Output channels

How to output more than just a stereo sound. Using the techniques talked about here an application can simultaneously output sound on as many channels as the Macintosh has.

Multiple Sound Input Channels

How to record more than just a stereo sound. Using the techniques talked about here, an application can simultaneously record sound via as many input sources as the Macintosh has.

Volume and Panning

How you can perform snazzy audio effects.

Monitoring a Sound Channel

How to use sound components to monitor the sound output level of a particular sound channel.

## Multiple Sound Output Channels

The Sound Manager has supported this starting with Sound Manager 3.0 which shipped with System 7.0. *Inside Macintosh: Sound* page 2-128 mentions this feature but doesn't describe how it is used.

When you call `SndNewChannel` one of the parameters passed is a long, which specifies the initialization parameters for the channel. Normally, developers just pass nil, or `initMono` or `initStereo`, but you can also pass `kUseOptionalOutputDevice` (which is defined to be -1).

The use of `kUseOptionalOutputDevice` will allow you to specify a different output component to have the sound played through. With `kUseOptionalOutputDevice`, a developer is able to play sounds simultaneously on as many output devices as are installed. For instance, a hardware developer who has a six-channel card could make three stereo output components (one for each pair of output channels) and the `kUseOptionalOutputDevice` selector would allow developers writing software to this card to play three stereo sounds simultaneously.

The `kUseOptionalOutputDevice` selector works like this: First, you have to find the component instance of the output component you wish to use (this example finds the AIFF Writer sample output component):

```
    //Find our output component's instance
    outputDev.componentType = 'sdev';
    outputDev.componentSubType = 'AIFW';
    outputDev.componentManufacturer = 'appl';
    outputDev.componentFlags = 0;
    outputDev.componentFlagsMask = 0;

    theAIFWComponent = FindNextComponent (0, &outputDev);
```

A generic routine to find all sound output components such as:

```
    long FindAllsdevs (Component ** componentsArray) {
    Component                   foundComponent,
                                aComponent;
    ComponentDescription        looking;
    long                        numComponents,
                                i;

    aComponent                      = 0;
    looking.componentType           = kSoundOutputDeviceType; // 'sdev'
    looking.componentSubType        = 0;
    looking.componentManufacturer   = 0;
    looking.componentFlags          = 0;
    looking.componentFlagsMask      = 0;

    numComponents = CountComponents (&looking);

    *componentsArray = (Component*)NewPtr (sizeof (Component) * numComponents);
    if (componentsArray == nil) {
        numComponents = 0;      // won't be able to list them anyway
    }

    for (i = 0; i < numComponents; i++) {
        foundComponent = FindNextComponent (aComponent, &looking);
        (*componentsArray)[i] = foundComponent;
        aComponent = foundComponent;    // continue looking
    }

    return numComponents;
}
```

would be used to present the user with a list of sound output channels so that they could output multi-channel (greater than 2 channel) sounds without the sound having to be mixed down to two channels.

Now, all that is left to do is to make a new sound channel which will use the selected ouput device:

```
err = SndNewChannel (&theOptionalOutputChan, kUseOptionalOutputDevice,
     (long)theAIFWComponent, nil);
```

This is all there is to it. Whenever a sound is played through `theOptionalOutputChan` the sound will not go to the built-in hardware: instead, it will go to the AIFF Writer output device (which could just as easily be the third and fourth channels of a multiple-channel output card).

So, what does a developer of multiple-channel hardware have to do to allow the use of the `theOptionalOutputChan` selector?

Not much. All that has to be done is to make an output component for each pair of output channels the hardware supports. If your hardware has only two channels, one output component is all that is needed. If your hardware has 20 output channels, 10 output components would be required.

I can almost hear developers now: "Ten output components?!? Are you crazy?!?"

No. Because of the reusability inherent to components, all that is really required is ten '`thng`' resources that register the same output component ten times as if it was ten different output components.

The output component should be written in such a way that the actual hardware channels it outputs to are abstracted: that way, one code base can talk to all available channels. The output component just looks at itself at register time to determine which output channels it controls and saves this information in its globals. The additional code required is very minor: just enough code to keep track of which channels to output on.

**Note:**
You can get the Speech Manager to talk through an optional output channel as well. Call `NewSpeechChannel` as you normally would and then call `SetSpeechInfo` with the `soSoundOutput` selector and the component instance of the output component:

```
err = SetSpeechInfo (theAIFWSpeechChan, soSoundOutput, &theAIFWComponent);
```

## Multiple Sound Input Channels

This works in much the same way as multiple sound output channels. The difference is, with sound input, a driver is required rather than a component.

Because an application can open as many sound input drivers as are available, all that an application must do is call `SPBOpenDevice` multiple times, each time with the name of each sound input drivers.

The Sound Manager provides a call which allows a developer to easily enumerate all the available sound input drivers.

```
SPBGetIndexedDevice (index, drvrName, &drvrIcon);
```

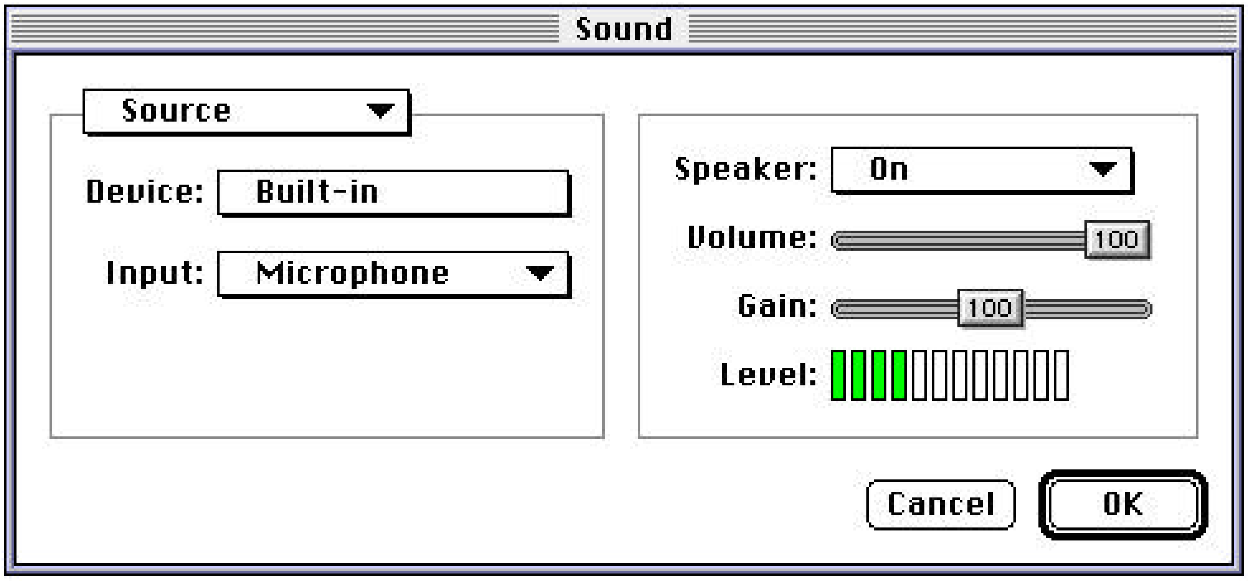
By pass an index starting at one (1) and incrementing it until an error is returned, you can quickly and easily build a list of all available sound input drivers.

## The QuickTime Way

You can also simply use, QuickTime to do your recording by using the Sequence Grabber to set the input source and do the recording.

This code will bring up the QuickTime sequence grabber sound input panel:

```
    ComponentResult     err;
    SGChannel           sgSoundChanRef;
    SeqGrabComponent    sgComponent;

    sgComponent = OpenDefaultComponent (SeqGrabComponentType, 0);

    err = SGInitialize (sgComponent);

    if (err == noErr) {
        err = SGNewChannel (sgComponent, SoundMediaType, &sgSoundChanRef);
    }

    if (err == noErr) {
        err = SGSettingsDialog (sgComponent, sgSoundChanRef, 0, nil, 0L, nil, nil);
    }

    return err;
}
```


Sound

Source

Device: Built-in

Input: Microphone

Speaker: On

Volume: 100

Gain: 100

Level:

One of the nice features of using QuickTime to record sounds is that QuickTime will rate: convert a sound for you, so that you can effectively record at any arbitrary sample rate, instead of being limited to recording at the sample rates that the specific sound input driver offers.

Making multiple sound input drivers for a hardware vendor is a little more work for the developer, who has to completely duplicate the driver: but that is what is required.

## Volume and Panning

Adjusting the volume of a sound playing through the Sound Manager can be done with `volumeCmd` issued with a `SndDoImmediate` call as this code demonstrates:

```
SndCommand      theCmd;
UInt16      rightVol, leftVol;

theCmd.cmd = volumeCmd;
theCmd.param2 = (rightVol << 16) | leftVol;
err = SndDoImmediate(chan, &theCmd);
```

The left and right volumes are actually 16-bit fixed point numbers. Like their 32-bit counterpart, the high 8 bits are the integer portion of the volume and the low 8 bits are the fractional portion of the volume. For example, a volume setting of `0x01000100` would be full volume on both channels, while a setting of `0x01000080` would be full volume on the right channel and half volume on the left channel.

A timed sequence of such calls with increasing values for the left volume and decreasing values for the right volume would make the sound pan from left to right.

Some developers may be asking, "What happens if I set the volume for a channel above `0x0100`?" The answer is: the sound gets louder. That's right, with `volumeCmd` you can overdrive the sound level for that loud crunchy sound so many rockers look for.

### The QuickTime Way

Another (probably better) way to control the volume and panning of sounds is to use QuickTime, The utility knife. Using QuickTime 2.1 modifiers track allows you to play sounds with complex effects.

### Tween Media handlers

Using the Tween Media handlers supplied by QuickTime 2.5, developers only need to specify the start and stop volume values for each sound channel; the Tween component generates all the intermediate volume values. This is in contrast with QuickTime 2.1, which did not have the Tween Media handlers; volume pans would have to be done with numerous discrete values. The Tween Media handlers simplify things by allowing the developer to simply specify start and stop values; the Tween Media handler takes care of coming up with the correct discrete value as the movie (sound) progresses.

For more information about how to use Tween Media handlers, see chapter 13 of the "Developer's Guide: QuickTime for Macintosh version 2.5".

Modifier tracks are talked about in "Developer's Guide: QuickTime for Macintosh version 2.5" starting on page 1-21.

## Monitoring a Sound Channel

Some developers wish to monitor sound output channels for various purposes, such as level metering. In the past, such an activity was very difficult because there is no easy way to get the Sound Manager's buffer; you had to guess where you were in the currently playing sound.

Sound Manager 3.2.1 helps to solve this problem by allowing users to install pre-mixer components. These are components that are installed in the component chain right before the Apple Mixer component.

A pre-mixer component sees the converted sound data from the channel it is installed on. That is, it sees the uncompressed, rate-converted, channel-converted, and size-converted data that the Apple Mixer is going to mix in with the other currently playing sounds.

Currently, there is no way to install a post-mixer component which would see the mixed result of all sound channels. Well, actually there is, they are called sound output components.

Writing a pre-mixer component is just like writing any other sound component see *Inside Macintosh: Sound* chapter 4 for the required selectors a sound component must support.

To install a pre-mixer component, you use a new `SPBSetDeviceInfo` selector, `siPreMixerSoundComponent` and pass a pointer to `SoundComponentLink` that describes the pre-mixer component you want installed.

This sample function shows how to create a simple sound channel with a specific pre-mixer component installed on that sound channel.

```
SndChannelPtr CreateChannelWithPreMixer (SndCallBackUPP callbackRoutine, OSType pmcSubTye)
{
    SoundComponentLink  preMixerCmp;
    SndChannelPtr       theChannel = nil;
    OSErr               err;

    /* create a new sound channel */
    err = SndNewChannel (&theChannel, sampledSynth, 0,
      callbackRoutine);

    if (err == noErr) {
        /* define the pre-mixer component */
        preMixerCmp.description.componentType = kSoundEffectsType;
        preMixerCmp.description.componentSubType = pmcSubTye;
        preMixerCmp.description.componentManufacturer = 0;
        preMixerCmp.description.componentFlags = 0;
        preMixerCmp.description.componentFlagsMask = 0;
        preMixerCmp.mixerID = nil;
        preMixerCmp.linkID = nil;
    }

    if (err == noErr) {
        /* install the pre-mixer component BEFORE the Apple Mixer */
        err = SndSetInfo (theChannel, siPreMixerSoundComponent, &preMixerCmp);
    }

    if (err != noErr) {
        theChannel = nil;
    }

    return (theChannel);
}
```

To send and receive information to and from your pre-mixer component, use the `SndSetInfo` and `SndGetInfo` functions, respectively. For example, this call could get the current value from your level meter component:

```
err = SndGetInfo (theChannel, LMValue, &level);
```

## Significant Restriction

There is one significant restriction on pre-mixer components -- they cannot increase the length of the sound. This is an important restriction if you happen to be writing a reverb or fade component. In order for these types of effects to work correctly, the sounds that are played must have long silent endings that the component can replace with its effect. A pre-mixer component can shorten the length of a sound, but it cannot increase it.

## Summary

These are some of the lesser known features of the Sound Manager. Now that you know them, I hope you will be able to take advantage of them and produce some of the best sounding applications on the planet.

## Further References

- *Inside Macintosh: Sound*
- The Sound Manager's web page
- Sound Manager addendum
- Technote 1048: Some Sound Advice: Getting the Most Out of the Sound Manager

## Downloadables

Acrobat version of this Note