This chapter describes sound components, which are code modules used by the Sound Manager to manipulate audio data or to communicate with sound output devices. Current versions of the Sound Manager allow you to write two kinds of sound components:

- compression and decompression components (codecs), which allow you to implement audio data compression and decompression algorithms different from those provided by the Sound Manager's MACE (Macintosh Audio Compression and Expansion) capabilities
- sound output device components, which send audio data directly to sound output devices

You need to read this chapter only if you are developing a sound output device or if you want to implement a custom compression and decompression scheme for audio data. For example, you might write a codec to handle 16-bit audio data compression and decompression. (The MACE algorithms currently compress and expand only 8-bit data at ratios of 3:1 and 6:1.)

IMPORTANT

Sound components are loaded and managed by the Sound Manager and operate transparently to applications. Applications that want to create sounds must use Sound Manager routines to do so. The routines described in this chapter are intended for use exclusively by sound components. ▲

To use this chapter, you should already be familiar with the general operation of the Sound Manager, as described in the chapter "Introduction to Sound on the Macintosh" in this book. Because sound components are components, you also need to be familiar with the Component Manager, described in *Inside Macintosh: More Macintosh Toolbox*. If you are developing a sound output device component, you need to be familiar with the process of installing a driver and handling interrupts created by your hardware device. See *Inside Macintosh: Devices* for complete information on devices and device drivers.

If you're developing a sound output device, you might also need to write a control panel extension that installs a custom subpanel into the Sound control panel. For example, your subpanel could allow the user to set various characteristics of the sound your output device is creating. For complete information on writing control panel subpanels, see the chapter "Control Panel Extensions" in *Inside Macintosh: Operating System Utilities*.

This chapter begins with a general description of sound components and how they are managed by the Sound Manager. Then it provides instructions on how to write a sound component. The section "Sound Components Reference" beginning on page 5-22 describes the sound component selectors your component might need to handle and the component-defined routines that your sound component should call in response to those the sound component selectors. It also describes a small number of Sound Manager utility routines that your sound component can use.

Note
Pascal interfaces for sound components are not currently available. As a result, this chapter provides all source code examples and reference materials in C. ◆

About Sound Components

A **sound component** is a component that works with the Sound Manager to manipulate audio data or to communicate with a sound output device. Sound components provide the foundation for the modular, device-independent sound architecture introduced with Sound Manager version 3.0. This section provides a description of sound components and shows how they are managed by the Sound Manager. For specific information on creating a sound component, see "Writing a Sound Component" beginning on page 5-8.

Sound Component Chains

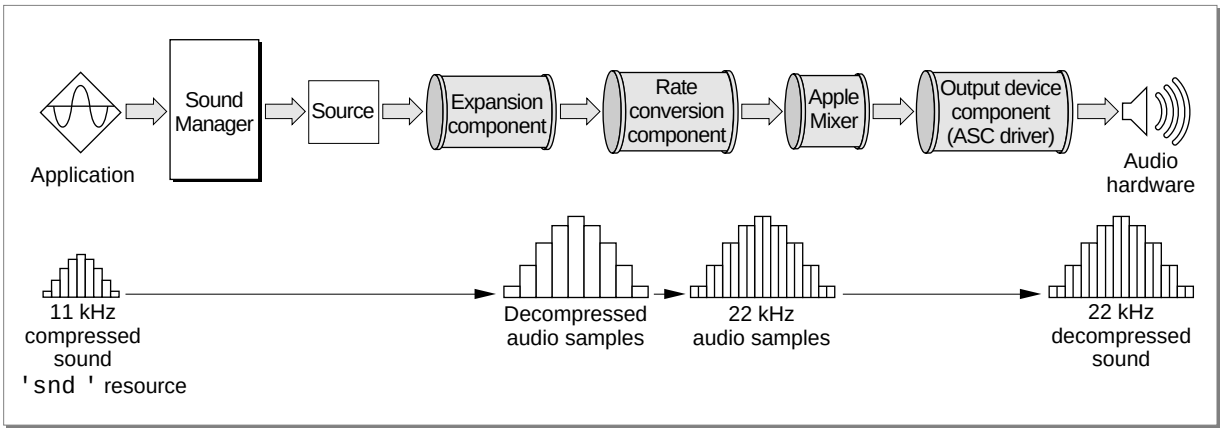
Prior to version 3.0, the Sound Manager performed all audio data processing internally, using its own filters to decompress audio data, convert sample rates, mix separate sound channels, and so forth. This effectively rendered it difficult, if not impossible, to add other data modification filters to process the audio data. (The now-obsolete method of installing a sound modifier with the `SndAddModifier` routine did not work reliably.) More importantly, the Sound Manager was responsible for managing the entire stream of audio data, from the application to the available sound-producing audio hardware. This made it very difficult to support new sound output devices.

In versions 3.0 and later, the Sound Manager provides a new audio data processing architecture based on components, illustrated in Figure 5-1. The fundamental idea is that the process of playing a sound can be divided into a number of specific steps, each of which has well-defined inputs and outputs. Figure 5-1 shows the steps involved in playing an 11 kHz compressed sampled sound resource on a Macintosh II computer.

An application sends the compressed sound data to the Sound Manager, which constructs an appropriate **sound component chain** that links the unprocessed audio data to the sound components required to modify the data into a form that can be sent to the current sound output device. As you can see in Figure 5-1, the Sound Manager links together sound components that, in sequence, expand the compressed sound data into audio samples, convert the sample rate from 11 kHz to 22 kHz, mix those samples with samples from any other sound channels that might be playing, and then write the samples to the available audio hardware (in this case, the FIFO buffer in the Apple Sound Chip).

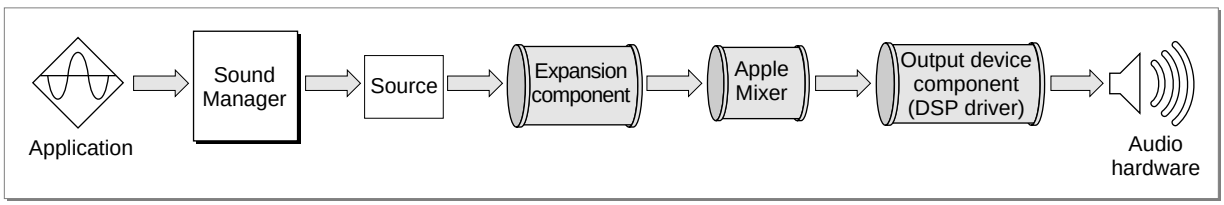
IMPORTANT
The Sound Manager itself converts both wave-table data and square-wave data into sampled-sound data before sending the data into a chain of sound components. As a result, sound components need to be concerned only with sampled-sound data. ▲

Figure 5-1 The component-based sound architecture



The components in a component chain may vary, depending both on the format of the audio data sent to the Sound Manager by an application and on the capabilities of the current sound output device. The chain shown in Figure 5-1 is necessary to handle the compressed 11 kHz sound because the Apple Sound Chip can handle only 22 kHz noncompressed sampled-sound data. Other sound output devices may be able to do more processing internally, thereby reducing the amount of processing required by the sound component chain. For instance, a DSP-based sound card might be capable of converting sample rates itself. In that case, the Sound Manager would not install the rate conversion component into the sound component chain. The resulting sound component chain is shown in Figure 5-2.

Figure 5-2 A component chain for audio hardware that can convert sample rates



The principal function of a sound component is to transfer data from the source down the chain of sound components while performing some specific modification on the data. It does this by getting a block of data from its **source component** (the component that immediately precedes it in the chain). The sound component then processes that data and stores it in the component's own private buffers. The next component can then get that processed data, perform its own modifications, and pass the data to the next component in the chain. Eventually, the audio data flows through the Apple Mixer (described in the next section) to the **sound output device component,** which sends the data to the current sound output device.

Notice that only the sound output device component communicates directly with the sound output hardware. This insulates all other sound components from having to know anything about the current sound output device. Rather, those components (sometimes called **utility components**) can simply operate on a stream of bytes.

The Sound Manager provides sound output device components for all sound output devices built into Macintosh computers. It also provides utility components for many typical kinds of audio data manipulation, including

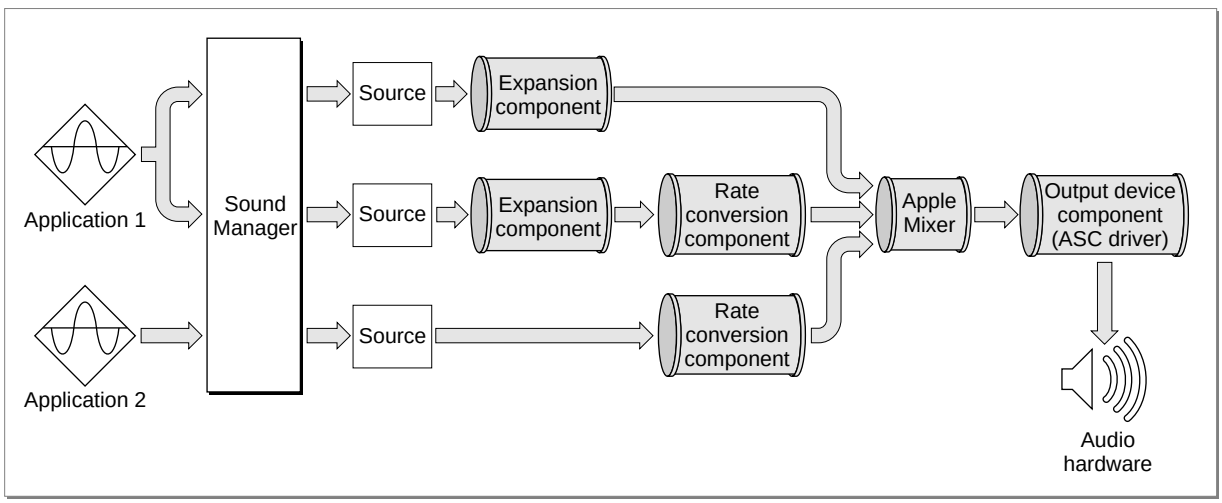
- sample rate conversion
- audio data expansion
- sample size conversion
- format conversion (for example, converting offset binary data to two's complement)

Currently, you can write sound output device components to handle communication with your own sound output devices. You can also write utility components to handle custom compression and expansion schemes. You cannot currently write any other kind of utility component.

The Apple Mixer

As you've seen, most sound components take a single source of audio data and modify it in some way, thereby producing a single output stream of audio data. There is one special sound component, known as the **Apple Mixer component** (or, more briefly, the **Apple Mixer**), that is able to handle more than one input data stream. Its function is precisely to mix together all open channels of sound data into a single output stream, as shown in Figure 5-3.

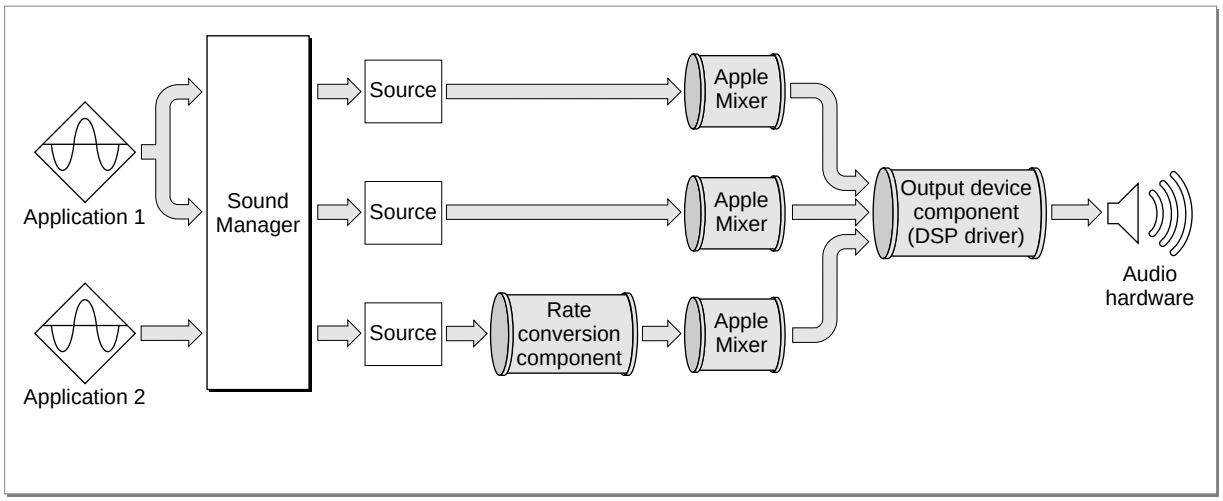
Figure 5-3 Mixing multiple channels of sound



The Apple Mixer has a more general function also, namely to construct the sound component chain required to process audio data from a given sound source into a format that can be handled by a particular sound output device. The Apple Mixer always feeds its output directly to the sound output device component, which sends the data to its associated audio hardware. After creating the component chain, the Apple Mixer assigns it a **source ID,** a 4-byte token that provides a unique reference to the component chain. The Apple Mixer is actually created by the sound output device component, when that component calls the Sound Manager's `OpenMixerSoundComponent` function.

In addition to creating sound component chains and mixing their data, the Apple Mixer can control the volume and stereo panning of a particular sound channel. Some sound output devices might be able to provide these capabilities as well. Indeed, some sound output devices might even be able to mix the data in multiple sound channels. In those cases, the sound output device component can call the `OpenMixerSoundComponent` function once for each sound source it wants to manage. The result is a separate instance of the Apple Mixer for each sound source, as shown in Figure 5-4.

Figure 5-4 A sound output device component that can mix sound channels



The sound output device component can instruct each instance of the Apple Mixer to pass all the sound data through unprocessed, thereby allowing the output device to perform the necessary processing and mixing. In this case, the Apple Mixer consumes virtually no processing time. The Apple Mixer must, however, still be present to set up the sound component chain and to assign a source ID to each sound source.

The Data Stream

A sound component is a standalone code resource that performs some signal processing function or communicates with a sound output device. All sound components have a standard programming interface and local storage that allows them to be connected

together in series to perform a wide range of audio data processing tasks. As previously indicated, all sound components (except for mixer components and some sound output device components) accept a single stream of input data and produce a single stream of output data.

The Sound Manager sends your sound component information about its input stream by passing it the address of a **sound component data record,** defined by the `SoundComponentData` data type.

```
typedef struct {
   long           flags;         /*sound component flags*/
   OSType         format;        /*data format*/
   short          numChannels;   /*number of channels in data*/
   short          sampleSize;    /*size of a sample*/
   UnsignedFixed  sampleRate;    /*sample rate*/
   long           sampleCount;   /*number of samples in buffer*/
   Byte           *buffer;       /*location of data*/
   long           reserved;      /*reserved*/
} SoundComponentData, *SoundComponentDataPtr;
```

The `buffer` field points to the buffer of input data. The other fields define the format of that data. For example, the sample size and rate are passed in the `sampleSize` and `sampleRate` fields, respectively. A utility component should modify the data in that buffer and then write the processed data into an internal buffer. Then it should fill out a sound component data record and pass its address back to the Sound Manager, which will then pass it on to the next sound component in the chain. Eventually, the audio data passes through all utility components in the chain, through the Apple Mixer and the sound output device component, down to the audio hardware.

Writing a Sound Component

A sound component is a component that works with the Sound Manager to manipulate audio data or to communicate with a sound output device. Because a sound component is a component, it must be able to respond to standard selectors sent by the Component Manager. In addition, a sound component must handle other selectors specific to sound components. This section describes how to write a sound component.

Creating a Sound Component

A sound component is a component. It contains a number of resources, including icons, strings, and the standard component resource (a resource of type `'thng'`) required of any Component Manager component. In addition, a sound component must contain code to handle required selectors passed to it by the Component Manager as well as selectors specific to the sound component.

Note
For complete details on components and their structure, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*. This section provides specific information about sound components. ◆

The component resource binds together all the relevant resources contained in a component; its structure is defined by the `ComponentResource` data type.

```
struct ComponentResource {
   ComponentDescription    cd;
   ResourceSpec            component;
   ResourceSpec            componentName
   ResourceSpec            componentInfo;
   ResourceSpec            componentIcon;
};
```

The `component` field specifies the resource type and resource ID of the component's executable code. By convention, this field should be set to the value `kSoundComponentCodeType`:

```
#define kSoundComponentCodeType     'sift'   /*sound component code type*/
```

(You can, however, specify some other resource type if you wish.) The resource ID can be any integer greater than or equal to 128. See the following section for further information about this code resource. The `ResourceSpec` data type has this structure:

```
typedef struct {
   OSType                  resType;
   short                   resID;
} ResourceSpec;
```

The `componentName` field specifies the resource type and resource ID of the resource that contains the component's name. Usually the name is contained in a resource of type `'STR '`. This string should be as short as possible.

The `componentInfo` field specifies the resource type and resource ID of the resource that contains a description of the component. Usually the description is contained in a resource of type `'STR '`.

The `componentIcon` field specifies the resource type and resource ID of the resource that contains an icon for the component. Usually the icon is contained in a resource of type `'ICON'`.

The `cd` field of the `ComponentResource` structure is a **component description record,** which contains additional information about the component. A component description record is defined by the `ComponentDescription` data type.

```
typedef struct {
   OSType                  componentType;
   OSType                  componentSubType;
```

```
   OSType                  componentManufacturer;
   unsigned long           componentFlags;
   unsigned long           componentFlagsMask;
} ComponentDescription;
```

For sound components, the `componentType` field must be set to a value recognized by the Sound Manager. Currently, there are five available component types for sound components:

```
#define kSoundComponentType      'sift'   /*utility component*/
#define kMixerType               'mixr'   /*mixer component*/
#define kSoundHardwareType       'sdev'   /*sound output device component*/
#define kSoundCompressor         'scom'   /*compression component*/
#define kSoundDecompressor       'sdec'   /*decompression component*/
```

In addition, the `componentSubType` field must be set to a value that indicates the type of audio services your component provides. For example, the Apple-supplied sound output device components have these subtypes:

```
#define kClassicSubType          'clas'   /*Classic hardware*/
#define kASCSubType              'asc '   /*ASC device*/
#define kDSPSubType              'dsp '   /*DSP device*/
```

If you add your own sound output device component, you should define some other subtype.

Note
Apple Computer, Inc., reserves for its own use all types and subtypes composed solely of lowercase letters. ◆

You can assign any value you like to the `componentManufacturer` field; typically you put the signature of your sound component in this field.

The `componentFlags` field of the component description for a sound component contains bit flags that encode information about the component. You can use this field to specify that the Component Manager should send your component the `kComponentRegisterSelect` selector.

```
enum {
   cmpWantsRegisterMessage    = 1L<<31 /*send register request*/
};
```

This bit is most useful for sound output device components, which might need to test for the presence of the appropriate hardware to determine whether to register with the Component Manager. When your component gets the `kComponentRegisterSelect` selector at system startup time, it should make sure that all the necessary hardware is available. If it isn't available, your component shouldn't register. See "Registering and Opening a Sound Component" beginning on page 5-16 for more information on opening and registering your sound component.

You also use the `componentFlags` field of the component description to define the characteristics of your component. For example, you can set a bit in that field to indicate that your sound component can accept stereo sound data. See "Specifying Sound Component Capabilities" on page 5-11 for more details on specifying the features of your sound component.

You should set the `componentFlagsMask` field to 0.

Listing 5-1 shows, in Rez format, a component resource for a sample sound output device component named SurfBoard.

Listing 5-1 Rez input for a component resource

```
#define kSurfBoardID                    128
#define kSurfBoardSubType               'SURF'
resource 'thng' (kSurfBoardID, purgeable) {
   'sdev',                      /*component type*/
   kSurfBoardSubType,           /*component subtype*/
   'appl',                      /*component manufacturer*/
   cmpWantsRegisterMessage,     /*component flags*/
   0,                           /*component flags mask*/
   'sift',                      /*component code resource type*/
   kSurfBoardID,                /*component code resource ID*/
   'STR ',                      /*component name resource type*/
   kSurfBoardID,                /*component name resource ID*/
   'STR ',                      /*component info resource type*/
   kSurfBoardID+1,              /*component info resource ID*/
   'ICON',                      /*component icon resource type*/
   kSurfBoardID                 /*component icon resource ID*/
};
```

Your sound component is contained in a resource file. You can assign any type you wish to be the file creator, but the type of the file must be `'thng'`. If the sound component contains a `'BNDL'` resource, then the file's bundle bit must be set.

Specifying Sound Component Capabilities

As mentioned in the previous section, the `componentFlags` field of a component description for a sound component contains bit flags that encode information about the component. The high-order 8 bits of that field are reserved for use by the Component Manager. In those 8 bits, you can set the `cmpWantsRegisterMessage` bit to indicate that the Component Manger should call your component during registration.

The low-order 24 bits of the `componentFlags` field of a component description are used by the Sound Manager. You'll set some of these bits to define the capabilities of

your sound component. You can use the following constants to set specific bits in the `componentFlags` field.

```
#define k8BitRawIn              (1 << 0)    /*data flags*/
#define k8BitTwosIn             (1 << 1)
#define k16BitIn                (1 << 2)
#define kStereoIn               (1 << 3)
#define k8BitRawOut             (1 << 8)
#define k8BitTwosOut            (1 << 9)
#define k16BitOut               (1 << 10)
#define kStereoOut              (1 << 11)
#define kReverse                (1 << 16)   /*action flags*/
#define kRateConvert            (1 << 17)
#define kCreateSoundSource      (1 << 18)
#define kHighQuality            (1 << 22)   /*performance flags*/
#define kRealTime               (1 << 23)
```

These constants define four types of information about your sound component: the kind of audio data it can accept as input, the kind of audio data it can produce as output, the actions it can perform on the audio data it's passed, and the performance of your sound component. For example, a utility component that accepts only monaural 8-bit, offset binary data as input and converts it to 16-bit two's complement data might have the value 0x00000801 (that is, `k8BitRawIn | k16BitOut`) in the `componentFlags` field.

The Sound Manager also defines a number of masks that you can use to select ranges of bits within the `componentFlags` field. See "Sound Component Features Flags" on page 5-26 for complete information on the defined bit constants and masks.

Dispatching to Sound Component-Defined Routines

As explained earlier, the code stored in the sound component should be contained in a resource of type `kSoundComponentCodeType`. The Component Manager expects the entry point in this resource to be a function with this format:

```
pascal ComponentResult MySurfDispatch (ComponentParameters *params,
                                       SoundComponentGlobalsPtr globals);
```

The Component Manager calls your sound component by passing `MySurfDispatch` a selector in the `params->what` field; `MySurfDispatch` must interpret the selector and possibly dispatch to some other routine in the resource. Your sound component must be able to handle the required selectors, defined by these constants:

```
#define kComponentOpenSelect            -1
#define kComponentCloseSelect           -2
#define kComponentCanDoSelect           -3
#define kComponentVersionSelect         -4
```

```
#define kComponentRegisterSelect       -5
#define kComponentTargetSelect         -6
#define kComponentUnregisterSelect     -7
```

Note
For complete details on required component selectors, see the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox*. ◆

In addition, your sound component must be able to respond to component-specific selectors. Some of these selectors must be handled by your component; if your component doesn't implement one of these selectors, it should return the `badComponentSelector` result code. Other selectors should be delegated up the component chain. This allows the Sound Manager to query a particular component chain by passing a selector to the first component in the chain. If your component does not implement a delegable selector, it should call the Component Manager routine `DelegateComponentCall` to delegate the selector to its source component. If your sound component does implement a particular delegable selector, it should perform the operation associated with it. The Sound Manager defines a constant to designate the delegable selectors.

```
/*first selector that can be delegated up the chain*/
#define kDelegatedSoundComponentSelectors       0x0100
```

The Sound Manager can pass these selectors to your sound component:

```
enum {
   /*the following calls cannot be delegated*/
   kSoundComponentInitOutputDeviceSelect     = 1,
   kSoundComponentSetSourceSelect,
   kSoundComponentGetSourceSelect,
   kSoundComponentGetSourceDataSelect,
   kSoundComponentSetOutputSelect,
   /*the following calls can be delegated*/
   kSoundComponentAddSourceSelect = kDelegatedSoundComponentSelectors + 1,
   kSoundComponentRemoveSourceSelect,
   kSoundComponentGetInfoSelect,
   kSoundComponentSetInfoSelect,
   kSoundComponentStartSourceSelect,
   kSoundComponentStopSourceSelect,
   kSoundComponentPauseSourceSelect,
   kSoundComponentPlaySourceBufferSelect
};
```

You can respond to these selectors by calling the Component Manager routine `CallComponentFunctionWithStorage` or by delegating the selector to your component's source component. Listing 5-2 illustrates how to define a sound component entry point routine.

Listing 5-2 Handling Component Manager selectors

```
pascal ComponentResult MySurfDispatch (ComponentParameters *params,
                                       SoundComponentGlobalsPtr globals)
{
   ComponentRoutine     myRoutine;
   ComponentResult      myResult;

   /*Get address of component-defined routine.*/
   myRoutine = MyGetComponentRoutine(params->what);

   if (myRoutine == nil)                    /*selector not implemented*/
      myResult = badComponentSelector;
   else if (myRoutine == kDelegateCall)     /*selector should be delegated*/
      myResult = DelegateComponentCall(params, globals->sourceComponent);
   else
      myResult = CallComponentFunctionWithStorage((Handle) globals, params,
                                          (ComponentRoutine) myRoutine);
   return (myResult);
}
```

As you can see, the `MySurfDispatch` function defined in Listing 5-2 simply retrieves the address of the appropriate component-defined routine, as determined by the `params->what` field. If the routine `MyGetComponentRoutine` returns `nil`, then `MySurfDispatch` itself returns the `badComponentSelector` result code. Otherwise, if the selector should be delegated, `MySurfDispatch` calls `DelegateComponentCall` to do so. Finally, if the selector hasn't yet been handled, the appropriate component-defined routine is executed via `CallComponentFunctionWithStorage`.

Listing 5-3 defines the function `MyGetComponentRoutine`.

Listing 5-3 Finding the address of a component-defined routine

```
ComponentRoutine MyGetComponentRoutine (short selector)
{
   void                 *myRoutine;

   if (selector < 0)
      switch (selector)          /*required component selectors*/
      {
         case kComponentRegisterSelect:
            myRoutine = MyRegisterSoundComponent;
            break;
         case kComponentVersionSelect:
```

```
            myRoutine = MySoundComponentVersion;
            break;
         case kComponentCanDoSelect:
            myRoutine = MySoundComponentCanDo;
            break;
         case kComponentCloseSelect:
            myRoutine = MyCloseSoundComponent;
            break;
         case kComponentOpenSelect:
            myRoutine = MyOpenSoundComponent;
            break;
         default:
            myRoutine = nil;     /*unknown selector, so fail*/
            break;
      }
   else if (selector < kDelegatedSoundComponentSelectors)
                           /*selectors that can't be delegated*/
      switch (selector)
      {
         case kSoundComponentInitOutputDeviceSelect:
            myRoutine = MySoundComponentInitOutputDevice;
            break;

         case kSoundComponentSetSourceSelect:
         case kSoundComponentGetSourceSelect:
         case kSoundComponentGetSourceDataSelect:
         case kSoundComponentSetOutputSelect:
         default:
            myRoutine = nil;     /*unknown selector, so fail*/
            break;
      }
   else                    /*selectors that can be delegated*/
      switch (selector)
      {
         case kSoundComponentStartSourceSelect:
            myRoutine = MySoundComponentStartSource;
            break;
         case kSoundComponentPlaySourceBufferSelect:
            myRoutine = MySoundComponentPlaySourceBuffer;
            break;
         case kSoundComponentGetInfoSelect:
            myRoutine = MySoundComponentGetInfo;
            break;
```

```
            case kSoundComponentSetInfoSelect:
               myRoutine = MySoundComponentSetInfo;
               break;
            case kSoundComponentAddSourceSelect:
            case kSoundComponentRemoveSourceSelect:
            case kSoundComponentStopSourceSelect:
            case kSoundComponentPauseSourceSelect:
            default:
               myRoutine = kDelegateCall;             /*delegate it*/
               break;
         }

      return (myRoutine);
}
```

In all likelihood, your component is loaded into the system heap, although it might be loaded into an application heap if memory is low in the system heap. You can call the Component Manager function `GetComponentInstanceA5` to determine the A5 value of the current application. If this function returns 0, your component is in the system heap; otherwise, your component is in an application's heap. Its location might affect how you allocate memory. For example, calling the `MoveHHi` routine on handles in the system heap has no result. Thus, you should either call the `ReserveMemSys` routine before calling `NewHandleSys` (so that the handle is allocated as low in the system heap as possible) or else just allocate a nonrelocatable block by calling the `NewPtrSys` routine.

If you need to access resources that are stored in your sound component, you can use `OpenComponentResFile` and `CloseComponentResFile`. `OpenComponentResFile` requires the `ComponentInstance` parameter supplied to your routine. You should not call Resource Manager routines such as `OpenResFile` or `CloseResFile`.

▲ **WARNING**
Do not leave any resource files open when your sound component is closed. Their maps will be left in the subheap when the subheap is freed, causing the Resource Manager to crash. ▲

The following sections illustrate how to define some of the sound component functions.

Registering and Opening a Sound Component

The Component Manager sends your component the `kComponentRegisterSelect` selector, usually at system startup time, to allow your component to determine whether it wants to register itself with the Component Manager. Utility components should always register themselves, so that the capabilities they provide will be available when needed. Sound output device components, however, should first check to see whether any necessary hardware is available before registering themselves. If the hardware they drive isn't available, there is no point in registering with the Component Manager.

The Component Manager sends your component the `kComponentOpenSelect` selector whenever the Sound Manager wants to open a connection to your component. In general, a sound output device component has only one connection made to it. A utility component, however, might have several instances, if the capabilities it provides are needed by more than one sound component chain. Your component should do as little as possible when opening up. It should allocate whatever global storage it needs to manage the connection and call `SetComponentInstanceStorage` so that the Component Manager can remember the location of that storage and pass it to all other component-defined routines.

As noted in the previous section, your component is probably loaded into the system heap. If so, you should also allocate any global storage in the system heap. If memory is tight, however, your component might be loaded into an application's heap (namely, the heap of the first application that plays sound). In that case, you should allocate any global variables you need in that heap. The Sound Manager ensures that other applications will not try to play sound while the component is in this application heap.

IMPORTANT

Your component is always sent the `kComponentOpenSelect` component selector before it is sent the `kComponentRegisterSelect` selector. As a result, you should not attempt to initialize or configure any associated hardware in response to `kComponentOpenSelect`. ▲

The Sound Manager sends the `kSoundComponentInitOutputDeviceSelect` selector specifically to allow a sound output device component to perform any hardware-related operations. Your component should initialize the hardware to some reasonable default values, create the Apple Mixer, and allocate any other memory that might be needed. Listing 5-4 shows one way to respond to the `kSoundComponentInitOutputDeviceSelect` selector.

Listing 5-4 Initializing an output device

```
static pascal ComponentResult MySoundComponentInitOutputDevice
                  (SoundComponentGlobalsPtr globals, long actions)
{
#pragma unused (actions)
   ComponentResult         myResult;

   /*Make sure we got our globals.*/
   if (globals->hwGlobals == nil)
      return (notEnoughHardwareErr);

   /*Set up the hardware.*/
   myResult = MySetupHardware(globals);
   if (myResult != noErr)
      return (myResult);
```

```
    /*Create an Apple Mixer.*/
    myResult = OpenMixerSoundComponent(&globals->thisComp, 0,
                                       &globals->sourceComponent);

    return (myResult);
}
```

The `MySoundComponentInitOutputDevice` function defined in Listing 5-4 simply retrieves the location of its global variables, configures the hardware by calling the `MySetupHardware` function, and then calls `OpenMixerSoundComponent` to create an instance of the Apple Mixer.

Finding and Changing Component Capabilities

All sound components take a stream of input data and produce a (usually different) stream of output data. The Sound Manager needs to know what operations your component can perform, so that it knows what other sound components might need to be linked together to play a particular sound on the available sound output device. It calls your component's `SoundComponentGetInfo` and `SoundComponentSetInfo` functions to get and set information about the capabilities and current settings of your sound component.

To specify the kind of information it wants to get or set, the Sound Manager passes your component a **sound component information selector.** If your component does not support a particular selector, if should pass the selector to the specified sound source. If your component does support the selector, it should either return the desired information directly or alter its settings as requested.

The sound component information selectors can specify any of a large number of audio capabilities or component settings. For example, the selector `siRateMultiplier` is passed to get or set the current output sample rate multiplier value.

Note
The Sound Manager uses many of the sound input device information selectors defined by the Sound Input Manager for communicating with sound input devices. See "Sound Input Manager" in this book for a description of the sound input device information selectors. A complete list of all sound component information selectors is provided in "Sound Component Information Selectors" beginning on page 5-22. ◆

Your component's `SoundComponentGetInfo` function has the following declaration:

```
pascal ComponentResult SoundComponentGetInfo (ComponentInstance ti,
                                    SoundSource sourceID, OSType selector,
                                    void *infoPtr);
```

The sound component information selector is passed in the `selector` parameter. The sound source is identified by the source ID passed in the `sourceID` parameter.

The `infoPtr` parameter specifies the location in memory of the information returned by `SoundComponentGetInfo`. If the information to be returned occupies four bytes or fewer, you can simply return the information in the location pointed to by that parameter. Otherwise, you should pass back in the `infoPtr` parameter a pointer to a record of type `SoundInfoList`, which contains an integer and a handle to an array of data items. In the second case, you'll need to allocate memory to hold the information you need to pass back. Listing 5-5 defines a component's `SoundComponentGetInfo` routine. It returns information to the Sound Manager about its capabilities and current settings.

Listing 5-5 Getting sound component information

```
static pascal ComponentResult MySoundComponentGetInfo
                     (SoundComponentGlobalsPtr globals, SoundSource sourceID,
                        OSType selector, void *infoPtr)
{
   HandleListPtr        listPtr;
   short                *sp, i;
   UnsignedFixed        *lp;
   Handle               h;
   HardwareGlobalsPtr   hwGlobals = globals->hwGlobals;
   ComponentResult      result = noErr;

   /*Make sure we got our global variables.*/
   if (hwGlobals == nil)
      return (notEnoughHardwareErr);

   switch (selector)
   {
      case siSampleSize:                  /*return current sample size*/
         *((short *) infoPtr) = hwGlobals->sampleSize;
         break;

      case siSampleSizeAvailable:         /*return sample sizes available*/
         h = NewHandle(sizeof(short) * kSampleSizesCount);
         if (h == nil)
            return (MemError());

         listPtr = (HandleListPtr) infoPtr;
         listPtr->count = 0;              /*num. sample sizes in handle*/
         listPtr->handle = h;             /*handle to be returned*/

         sp = (short *) *h;               /*store sample sizes in handle*/
```

```
   for (i = 0; i < kSampleSizesCount; ++i)
      if (hwGlobals->sampleSizesActive[i])
      {
         listPtr->count++;
         *sp++ = hwGlobals->sampleSizes[i];
      }
   break;

case siSampleRate:                  /*return current sample rate*/
   *((Fixed *) infoPtr) = hwGlobals->sampleRate;
   break;

case siSampleRateAvailable:         /*return sample rates available*/
   h = NewHandle(sizeof(UnsignedFixed) * kSampleRatesCount);
   if (h == nil)
      return (MemError());

   listPtr = (HandleListPtr) infoPtr;
   listPtr->count = 0;              /*num. sample rates in handle*/
   listPtr->handle = h;             /*handle to be returned*/

   lp = (UnsignedFixed *) *h;

   /*If the hardware can support a range of sample rate values,
     the list count should be set to 0 and the minimum and maximum
     sample rate values should be stored in the handle.*/
   if (hwGlobals->supportsRateRange)
   {
      *lp++ = hwGlobals->sampleRateMin;
      *lp++ = hwGlobals->sampleRateMax;
   }

   /*If the hardware supports a limited set of sample rates,
     the list count should be set to the number of sample rates
     and this list of rates should be stored in the handle.*/
   else
   {
      for (i = 0; i < kSampleRatesCount; ++i)
         if (hwGlobals->sampleRatesActive[i])
         {
            listPtr->count++;
            *lp++ = hwGlobals->sampleRates[i];
```

```
            }
         }
         break;

      case siNumberChannels:              /*return current num. channels*/
         *((short *) infoPtr) = hwGlobals->numChannels;
         break;

      case siChannelAvailable:            /*return channels available*/
         h = NewHandle(sizeof(short) * kChannelsCount);
         if (h == nil)
            return (MemError());

         listPtr = (HandleListPtr) infoPtr;
         listPtr->count = 0;              /*num. channels in handle*/
         listPtr->handle = h;             /*handle to be returned*/

         sp = (short *) *h;               /*store channels in handle*/

         for (i = 0; i < kChannelsCount; ++i)
            if (hwGlobals->channelsActive[i])
            {
               listPtr->count++;
               *sp++ = hwGlobals->channels[i];
            }
         break;

      case siHardwareVolume:
         *((long *)infoPtr) = hwGlobals->volume;
         break;

      /*If you do not handle a selector, delegate it up the chain.*/
      default:
         result = SoundComponentGetInfo(globals->sourceComponent, sourceID,
                                        selector, infoPtr);
         break;
   }
   return (result);
}
```

You can define your `MySoundComponentSetInfo` routine in an exactly similar fashion.

Sound Components Reference

This section describes the constants, data structures, and routines you can use to write a sound component. It also describes the routines that your sound component should call in response to a sound component selector. See "Writing a Sound Component" on page 5-8 for information on creating a component that contains these component-defined routines.

Constants

This section provides details on the constants defined by the Sound Manager for use with sound components. You'll use these constants to

- determine the kind of information the Sound Manager wants your sound component to return to it or settings it wants your sound component to change
- define the format of the audio data your sound component is currently producing
- specify the action flags for the `SoundComponentPlaySourceBuffer` function
- specify the format of the data your sound output device component expects to receive

Sound Component Information Selectors

The Sound Manager calls your sound component's `SoundComponentGetInfo` and `SoundComponentSetInfo` functions to determine the capabilities of your component and to change those capabilities. It passes those functions a sound component information selector in the function's `selector` parameter to specify the type of information it wants to get or set. The available sound component information selectors are defined by constants.

Note
Most of these selectors can be passed to both `SoundComponentGetInfo` and `SoundComponentSetInfo`. Some of them, however, can be sent to only one or the other. ◆

```
#define siChannelAvailable       'chav'   /*number of channels available*/
#define siCompressionAvailable   'cmav'   /*compression types available*/
#define siCompressionFactor      'cmfa'   /*current compression factor*/
#define siCompressionType        'comp'   /*current compression type*/
#define siHardwareMute           'hmut'   /*current hardware mute state*/
#define siHardwareVolume         'hvol'   /*current hardware volume*/
#define siHardwareVolumeSteps    'hstp'   /*number of hardware volume steps*/
#define siHeadphoneMute          'pmut'   /*current headphone mute state*/
#define siHeadphoneVolume        'pvol'   /*current headphone volume*/
```

```
#define siHeadphoneVolumeSteps   'hdst'   /*num. of headphone volume steps*/
#define siNumberChannels         'chan'   /*current number of channels*/
#define siQuality                'qual'   /*current quality*/
#define siRateMultiplier         'rmul'   /*current rate multiplier*/
#define siSampleRate             'srat'   /*current sample rate*/
#define siSampleRateAvailable    'srav'   /*sample rates available*/
#define siSampleSize             'ssiz'   /*current sample size*/
#define siSampleSizeAvailable    'ssav'   /*sample sizes available*/
#define siSpeakerMute            'smut'   /*current speaker mute*/
#define siSpeakerVolume          'svol'   /*current speaker volume*/
#define siVolume                 'volu'   /*current volume setting*/
```

Constant descriptions

`siChannelAvailable`
: Get the maximum number of channels this sound component can manage, as well as the channels themselves. The `infoPtr` parameter points to a record of type `SoundInfoList`, which contains an integer (the number of available channels) and a handle to an array of integers (which represent the channel numbers themselves).

`siCompressionAvailable`
: Get the number and list of compression types this sound component can manage. The `infoPtr` parameter points to a record of type `SoundInfoList`, which contains the number of compression types, followed by a handle that references a list of compression types, each of type `OSType`.

`siCompressionFactor`
: Get information about the current compression type. The `infoData` parameter points to a compression information record (see page 5-32).

`siCompressionType`
: Get or set the current compression type. The `infoPtr` parameter points to a buffer of type `OSType`, which is the compression type.

`siHardwareMute`
: Get or set the current mute state of the audio hardware. A value of 0 indicates that the hardware is not muted, and a value of 1 indicates that the hardware is muted. Not all sound components need to support this selector; it's intended for sound output device components whose associated hardware can be muted.

`siHardwareVolume`
: Get or set the current volume level of all sounds produced on the sound output device. The `infoPtr` parameter points to a long integer, where the high-order word represents the right volume level and the low-order word represents the left volume level. A volume level is specified by an unsigned 16-bit number: 0x0000 represents silence and 0x0100 represents full volume. (You can use the constant `kFullVolume` for full volume.) You can specify values

larger than 0x0100 to overdrive the volume, although doing so might result in clipping. This selector applies to the volume of the output device, whereas the `siVolume` selector applies to the volume of a specific sound channel and its component chain. If a sound output device supports more than one output port (for example, both headphones and speakers), the `siHardwareVolume` selector applies to all those ports.

`siHardwareVolumeSteps`
Get the number of audible volume levels supported by the audio hardware. If the device supports a range of volume levels (for example, 0x000 to 0x1000), you should return only the number of levels that are audible. The Sound Manager uses this information to handle the volume slider in the Alert Sounds control panel.

`siHeadphoneMute`
Get or set the current mute state of the headphone. A value of 0 indicates that the headphone is not muted, and a value of 1 indicates that the headphone is muted. Not all sound components need to support this selector; it's intended for sound output device components whose associated headphone can be muted.

`siHeadphoneVolume`
Get or set the current volume level of all sounds produced on the headphone. The `infoPtr` parameter points to a long integer, where the high-order word represents the right volume level and the low-order word represents the left volume level. A volume level is specified by an unsigned 16-bit number: 0x0000 represents silence and 0x0100 represents full volume. (You can use the constant `kFullVolume` for full volume.) You can specify values larger than 0x0100 to overdrive the volume, although doing so might result in clipping. This selector applies to the volume of the headphones.

`siHeadphoneVolumeSteps`
Get the number of audible volume levels supported by the headphones. If the headphones support a range of volume levels (for example, 0x000 to 0x1000), you should return only the number of levels that are audible.

`siNumberChannels`
Get or set the current number of audio channels currently being managed by the sound component. The `infoPtr` parameter points to an integer, which is the number of channels. For example, for stereo sounds, this integer should be 2.

`siQuality` Get or set the current quality setting for the sound component. The `infoPtr` parameter points to a 32-bit value, which typically determines how much processing should be applied to the audio data stream.

`siRateMultiplier`
Get or set the current rate multiplier for the sound component. The `infoPtr` parameter points to a buffer of type `UnsignedFixed`, which is the multiplier to be applied to the playback rate of the sound, independent of the base sample rate of the sound. For example, if the current rate multiplier is 2.0, the sound is played

back at twice the speed specified in the `sampleRate` field of the sound component data record.

`siSampleRate` Get or set the current sample rate of the data being output by the sound component. The `infoPtr` parameter points to a buffer of type `UnsignedFixed`, which is the sample rate.

`siSampleRateAvailable`
Get the range of sample rates this sound component can handle. The `infoPtr` parameter points to a record of type `SoundInfoList`, which is the number of sample rates the component supports, followed by a handle to a list of sample rates, each of type `UnsignedFixed`. The sample rates can be in the range 0 to 65535.65535. If the number of sample rates is 0, then the first two sample rates in the list define the lowest and highest values in a continuous range of sample rates.

`siSampleSize` Get or set the current sample size of the audio data being output by the sound component. The `infoPtr` parameter points to an integer, which is the sample size in bits.

`siSampleSizeAvailable`
Get the range of sample sizes this sound component can handle. The `infoPtr` parameter points to a record of type `SoundInfoList`, which is the number of sample sizes the sound component supports, followed by a handle. The handle references a list of sample sizes, each of type `Integer`. Sample sizes are specified in bits.

`siSpeakerMute`
Get or set the current mute state of the speakers. A value of 0 indicates that the speakers are not muted, and a value of 1 indicates that the speakers are muted. Not all sound components need to support this selector; it's intended for sound output device components whose associated speakers can be muted.

`siSpeakerVolume`
Get or set the current volume level of all sounds produced on the speakers. The `infoPtr` parameter points to a long integer, where the high-order word represents the right volume level and the low-order word represents the left volume level. A volume level is specified by an unsigned 16-bit number: 0x0000 represents silence and 0x0100 represents full volume. (You can use the constant `kFullVolume` for full volume.) You can specify values larger than 0x0100 to overdrive the volume, although doing so might result in clipping. This selector applies to the volume of the speakers.

`siVolume` Get or set the current volume level of the sound component. The `infoPtr` parameter points to a long integer, where the high-order word represents the right volume level and the low-order word represents the left volume level. A volume level is specified by an unsigned 16-bit number: 0x0000 represents silence and 0x0100 represents full volume. (You can use the constant `kFullVolume` for full volume.) You can specify values larger than 0x0100 to overdrive the volume, although doing so might result in clipping. This selector applies to the volume of a specific sound channel and its

component chain, while the `siHardwareVolume` selector applies to the volume of the output device.

Audio Data Types

You can use the following constants to define the format of the audio data your sound component is currently producing. You can also define additional data types to denote your own compression schemes. You pass these constants in the `format` field of a sound component data record.

```
#define kOffsetBinary         'raw '
#define kTwosComplement       'twos'
#define kMACE3Compression     'MAC3'
#define kMACE6Compression     'MAC6'
```

Constant descriptions

`kOffsetBinary` The data is noncompressed samples in offset binary format (that is, values range from 0 to 255).

`kTwosComplement`
The data is noncompressed samples in two's complement format (that is, values range from –128 to 128).

`kMACE3Compression`
The data is compressed using MACE 3:1 compression.

`kMACE6Compression`
The data is compressed using MACE 6:1 compression.

Sound Component Features Flags

You can use the following constants to define features of your sound component. You use some combination of these constants to set bits in the `componentFlags` field of a component description record, which is contained in a `'thng'` resource. These bits represent the kind of data your component can receive as input, the kind of data your component can produce as output, the operations your component can perform, and the performance of your component.

```
#define k8BitRawIn              (1 << 0)    /*data flags*/
#define k8BitTwosIn             (1 << 1)
#define k16BitIn                (1 << 2)
#define kStereoIn               (1 << 3)
#define k8BitRawOut             (1 << 8)
#define k8BitTwosOut            (1 << 9)
#define k16BitOut               (1 << 10)
#define kStereoOut              (1 << 11)
#define kReverse                (1 << 16)   /*action flags*/
#define kRateConvert            (1 << 17)
```

```
#define kCreateSoundSource      (1 << 18)
#define kHighQuality            (1 << 22)   /*performance flags*/
#define kRealTime               (1 << 23)
```

Constant descriptions

`k8BitRawIn` The component can accept 8 bit offset binary data as input.

`k8BitTwosIn` The component can accept 8 bit two's complement data as input.

`k16BitIn` The component can accept 16 bit data as input. 16 bit data is always in two's complement format.

`kStereoIn` The component can accept stereo data as input.

`k8BitRawOut` The component can produce 8 bit offset binary data as output.

`k8BitTwosOut` The component can produce 8 bit two's complement data as output.

`k16BitOut` The component can produce 16 bit data as output. 16 bit data is always in two's complement format.

`kStereoOut` The component can produce stereo data as output.

`kReverse` The component can accept reversed audio data.

`kRateConvert` The component can convert sample rates.

`kCreateSoundSource` The component can create sound sources.

`kHighQuality` The component can produce high quality output.

`kRealTime` The component can operate in real time.

Action Flags

You can use constants to specify the action flags in the `actions` parameter of the `SoundComponentPlaySourceBuffer` function. See page 5-49 for information about this function.

```
#define kSourcePaused           (1 << 0)
#define kPassThrough            (1 << 16)
#define kNoSoundComponentChain  (1 << 17)
```

Constant descriptions

`kSourcePaused` If this bit is set, the component chain is configured to play the specified sound but the playback is initially paused. In this case, your `SoundComponentStartSource` function must be called to begin playback. If this bit is clear, the playback begins immediately once the component chain is set up and configured.

`kPassThrough` If this bit is set, the Sound Manager passes all data through to the sound output device component unmodified. A sound output device component that can handle any sample rate and sound format described in a sound parameter block should set this bit.

`kNoSoundComponentChain` If this bit is set, the Sound Manager does not construct a component chain for processing the sound data.

Data Format Flags

You can use constants to set or clear flag bits in the `outputFlags` parameter passed to the `OpenMixerSoundComponent` routine. These flags specify the format of the data your sound output device component expects to receive. See page 5-33 for information about the `OpenMixerSoundComponent` function.

IMPORTANT

Most of these flags are ignored unless the `kNoMixing` flag is set, because a sound output device component cannot perform data modifications such as sample rate conversion or sample size conversion unless it is also able to mix sound sources. ▲

```
#define kNoMixing                (1 << 0)    /*don't mix sources*/
#define kNoSampleRateConversion  (1 << 1)    /*don't convert sample rate*/
#define kNoSampleSizeConversion  (1 << 2)    /*don't convert sample size*/
#define kNoSampleFormatConversion         \
                                 (1 << 3)    /*don't convert sample format*/
#define kNoChannelConversion     (1 << 4)    /*don't convert stereo/mono*/
#define kNoDecompression         (1 << 5)    /*don't decompress*/
#define kNoVolumeConversion      (1 << 6)    /*don't apply volume*/
#define kNoRealtimeProcessing    (1 << 7)    /*don't run at interrupt time*/
```

Constant descriptions

`kNoMixing` If this bit is set, the Apple Mixer does not mix audio data sources.

`kNoSampleRateConversion`
If this bit is set, the sound component chain does not perform sample rate conversion (for example, converting 11 kHz data to 22 kHz data).

`kNoSampleSizeConversion`
If this bit is set, the sound component chain does not perform sample size conversion (for example, converting 8-bit data to 16-bit data).

`kNoSampleFormatConversion`
If this bit is set, the sound component chain does not convert between sample formats (for example, converting from two's complement data to offset binary data). Most sound output devices on Macintosh computers accept only 8-bit offset binary data, which is therefore the default type of data produced by the Apple Mixer. If your output device can handle either offset binary or two's complement data, you should set this flag. Note that 16-bit data is always in two's complement format.

`kNoChannelConversion`
If this bit is set, the sound component chain does not convert channels (for example, converting monophonic channels to stereo or stereo channels to monophonic).

`kNoDecompression`
If this bit is set, the sound component chain does not decompress

audio data. If your output device can decompress data, you should set this flag.

`kNoVolumeConversion`
If this bit is set, the sound component chain does not convert volumes.

`kNoRealtimeProcessing`
If this bit is set, the sound component chain does not do any processing at interrupt time.

Data Structures

This section describes the data structures you need to use when writing a sound component.

Sound Component Data Records

The flow of data from one sound component to another is managed using a sound component data record. This record indicates to other sound components the format of the data that a particular component is generating, together with the location and length of the buffer containing that data. This allows other sound components to access data from that component as needed. A sound component data record is defined by the `SoundComponentData` data type.

```
typedef struct {
   long           flags;              /*sound component flags*/
   OSType         format;             /*data format*/
   short          numChannels;        /*number of channels in data*/
   short          sampleSize;         /*size of a sample*/
   UnsignedFixed  sampleRate;         /*sample rate*/
   long           sampleCount;        /*number of samples in buffer*/
   Byte           *buffer;            /*location of data*/
   long           reserved;           /*reserved*/
} SoundComponentData, *SoundComponentDataPtr;
```

Field descriptions

`flags` A set of bit flags whose meanings are specific to a particular sound component.

`format` The format of the data a sound component is producing. The following formats are defined by Apple:

```
#define kOffsetBinary         'raw '
#define kTwosComplement       'twos'
#define kMACE3Compression     'MAC3'
#define kMACE6Compression     'MAC6'
```

See "Audio Data Types" on page 5-26 for a description of these formats. You can define additional format types, which are currently assumed to be the types of proprietary compression algorithms.

`numChannels` The number of channels of sound in the output data stream. If this field contains the value 1, the data is monophonic. If this field contains 2, the data is stereophonic. Stereo data is stored as interleaved samples, in a left-to-right ordering.

`sampleSize` The size, in bits, of each sample in the output data stream. Typically this field contains the values 8 or 16. For compressed sound data, this field indicates the size of the samples after the data has been expanded.

`sampleRate` The sample rate for the audio data. The sample rate is expressed as an unsigned, fixed-point number in the range 0 to 65536.0 samples per second.

`sampleCount` The number of samples in the buffer pointed to by the `buffer` field. For compressed sounds, this field indicates the number of compressed samples in the sound, not the size of the buffer.

`buffer` The location of the buffer that contains the sound data.

`reserved` Reserved for future use. You should set this field to 0.

Sound Parameter Blocks

The Sound Manager passes a component's `SoundComponentPlaySourceBuffer` function a **sound parameter block** that describes the source data to be modified or sent to a sound output device. A sound parameter block is defined by the `SoundParamBlock` data type.

```
struct SoundParamBlock {
   long                recordSize;    /*size of this record in bytes*/
   SoundComponentData  desc;          /*description of sound buffer*/
   Fixed               rateMultiplier;/*rate multiplier*/
   short               leftVolume;    /*volume on left channel*/
   short               rightVolume;   /*volume on right channel*/
   long                quality;       /*quality*/
   ComponentInstance   filter;        /*filter*/
   SoundParamProcPtr   moreRtn;       /*routine to call to get more data*/
   SoundParamProcPtr   completionRtn; /*buffer complete routine*/
   long                refCon;        /*user refcon*/
   short               result;        /*result*/
};
typedef struct SoundParamBlock SoundParamBlock;
typedef SoundParamBlock *SoundParamBlockPtr;
```

Field descriptions

`recordSize` The length, in bytes, of the sound parameter block.

`desc`	A sound component data record that describes the format, size, and location of the sound data. See "Sound Component Data Records" on page 5-29 for a description of the sound component data record.
`rateMultiplier`	A multiplier to be applied to the playback rate of the sound. This field contains an unsigned fixed-point number. If, for example, this field has the value 2.0, the sound is played back at twice the rate specified in the `sampleRate` field of the sound component data record contained in the `desc` field.
`leftVolume`	The playback volume for the left channel. You specify a volume with 16-bit value, where 0 (hexadecimal 0x0000) represents no volume and 256 (hexadecimal 0x0100) represents full volume. You can overdrive a channel's volume by passing volume levels greater than 0x0100.
`rightVolume`	The playback volume for the right channel. You specify a volume with 16-bit value, where 0 (hexadecimal 0x0000) represents no volume and 256 (hexadecimal 0x0100) represents full volume. You can overdrive a channel's volume by passing volume levels greater than 0x0100.
`quality`	The level of quality for the sound. This value usually determines how much processing should be applied during audio data processing (such as rate conversion and decompression) to increase the output quality of the sound.
`filter`	Reserved for future use. You should set this field to `nil`.
`moreRtn`	A pointer to a callback routine that is called to retrieve another buffer of audio data. This field is used internally by the Sound Manager.
`completionRtn`	A pointer to a callback routine that is called when the sound has finished playing. This field is used internally by the Sound Manager.
`refCon`	A value that is to be passed to the callback routines specified in the `moreRtn` and `completionRtn` fields. You can use this field to pass information (for example, the address of a structure) to a callback routine.
`result`	The status of the sound that is playing. The value 1 indicates that the sound is currently playing. The value 0 indicates that the sound has finished playing. Any negative value indicates that some error has occurred.

Sound Information Lists

The `SoundComponentGetInfo` and `SoundComponentSetInfo` functions access information about a sound component using a **sound information list,** which is defined by the `SoundInfoList` data type.

```
typedef struct {
   short          count;
   Handle         handle;
} SoundInfoList, *SoundInfoListPtr;
```

Field descriptions

`count` The number of elements in the array referenced by the `handle` field.

`handle` A handle to an array of data elements. The type of these data elements depends on the kind of information requested, which is determined by the `selector` parameter passed to `SoundComponentGetInfo` or `SoundComponentSetInfo`. See "Sound Component Information Selectors" beginning on page 5-22 for information about the available information selectors.

Compression Information Records

When the Sound Manager calls your `SoundComponentGetInfo` routine with the `siCompressionFactor` selector, you need to return a pointer to a **compression information record,** which is defined by the `CompressionInfo` data type.

```
typedef struct {
   long           recordSize;
   OSType         format;
   short          compressionID;
   short          samplesPerPacket;
   short          bytesPerPacket;
   short          bytesPerFrame;
   short          bytesPerSample;
   short          futureUse1;
} CompressionInfo, *CompressionInfoPtr, **CompressionInfoHandle;
```

Field descriptions

`recordSize` The size of this compression information record.

`format` The compression format.

`compressionID` The compression ID.

`samplesPerPacket`
The number of samples in each packet.

`bytesPerPacket`
The number of bytes in each packet.

`bytesPerFrame`
The number of bytes in each frame.

`bytesPerSample`
The number of bytes in each sample.

`futureUse1` Reserved for use by Apple Computer, Inc. You should set this field to 0.

Sound Manager Utilities

This section describes several utility routines provided by the Sound Manager that are intended for use only by sound components. You can use these routines to

- open and close the Apple Mixer component
- save and restore a user's preference settings for a sound component

Note
For a description of the routines that a sound component must implement, see "Sound Component-Defined Routines" on page 5-36. ◆

Opening and Closing the Apple Mixer Component

A sound output device component needs to open and close one or more instances of the Apple Mixer component.

OpenMixerSoundComponent

A sound output device component can use the `OpenMixerSoundComponent` function to open and connect itself to the Apple Mixer component.

```
pascal OSErr OpenMixerSoundComponent
                        (SoundComponentDataPtr outputDescription,
                        long outputFlags,
                        ComponentInstance *mixerComponent);
```

`outputDescription`
A description of the data format your sound output device is expecting to receive.

`outputFlags`
A set of 32 bit flags that provide additional information about the data format your output device is expecting to receive. See "Data Format Flags" beginning on page 5-28 for a description of the constants you can use to select bits in this parameter.

`mixerComponent`
The component instance of the Apple Mixer component. You need this instance to call the `SoundComponentGetSourceData` and `CloseMixerSoundComponent` functions.

DESCRIPTION

The `OpenMixerSoundComponent` function opens the standard Apple Mixer component and creates a connection between your sound output device component and the Apple Mixer. If your output device can perform specific operations on the

stream of audio data, such as channel mixing and rate conversion, it should call `OpenMixerSoundComponent` as many times as are necessary to create a unique component chain for each sound source. If, on the other hand, your output device does not perform channel mixing, it should call `OpenMixerSoundComponent` only once, from its `SoundComponentInitOutputDevice` function. This opens a single instance of the Apple Mixer component, which in turn manages all the available sound sources.

Your component specifies the format of the data it can handle by filling in a sound component data record and passing its address in the `outputDescription` parameter. The sound component data record specifies the data format as well as the sample rate and sample size expected by the output device component. If these specifications are sufficient to determine the kind of data your component can handle, you should pass the value 0 in the `outputFlags` parameter. Otherwise, you can set flags in the `outputFlags` parameter to select certain kinds of input data. For example, you can set the `kNoChannelConversion` flag to prevent the component chain from converting monophonic sound to stereo sound, or stereo sound to monophonic sound. See "Data Format Flags" beginning on page 5-28 for a description of the constants you can use to select bits in the `outputFlags` parameter.

SPECIAL CONSIDERATIONS

The `OpenMixerSoundComponent` function is available only in versions 3.0 and later of the Sound Manager. It should be called only by sound output device components.

CloseMixerSoundComponent

A sound output device component can use the `CloseMixerSoundComponent` function to close the Apple Mixer.

```
pascal OSErr CloseMixerSoundComponent (ComponentInstance ci);
```

`ci` The component instance of the Apple Mixer component.

DESCRIPTION

The `CloseMixerSoundComponent` function closes the Apple Mixer component instance specified by the `ci` parameter. Your output device component should call this function when it is being closed.

SPECIAL CONSIDERATIONS

The `CloseMixerSoundComponent` function is available only in versions 3.0 and later of the Sound Manager. It should be called only by sound output device components.

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	Invalid component ID

Saving and Restoring Sound Component Preferences

A sound component can use the `SetSoundPreference` and `GetSoundPreference` functions to save and restore a user's preference settings.

SetSoundPreference

A sound component can use the `SetSoundPreference` function to have the Sound Manager store a block of preferences data in a resource file. You're most likely to use this function in a sound output device component, although other types of sound components can use it also.

```
pascal OSErr SetSoundPreference (OSType type, Str255 name,
                                 Handle settings);
```

`type`	The resource type to be used to create the preferences resource.
`name`	The resource name to be used to create the preferences resource.
`settings`	A handle to the data to be stored in the preferences resource.

DESCRIPTION

The `SetSoundPreference` function causes the Sound Manager to attempt to create a new resource that contains preferences data for your sound component. You can use this function to maintain a structure of any format across subsequent startups of the machine. You'll retrieve the preferences data by calling the `GetSoundPreference` function. The data is stored in a resource with the specified type and name in a resource file in the Preferences folder in the System Folder. In general, the resource type and name should be the same as the sound component subtype and name.

The `settings` parameter is a handle to the preferences data you want to store. It is the responsibility of your component to allocate and initialize the block of data referenced by that handle. The Sound Manager copies the handle's data into a resource in the appropriate location. Your sound component should dispose of the handle when `SetSoundPreference` returns.

The format of the block of preferences data referenced by the `settings` parameter is defined by your sound component. It is recommended that you include a field specifying the version of the data format; this allows you to modify the format of the block of data while remaining compatible with previous formats you might have defined.

SPECIAL CONSIDERATIONS

The `SetSoundPreference` function is available only in versions 3.0 and later of the Sound Manager.

GetSoundPreference

A sound component can use the `GetSoundPreference` function to have the Sound Manager read a block of preferences data from a resource file. You'll use it to retrieve a block of preferences data you previously saved by calling `SetSoundPreference`.

```
pascal OSErr GetSoundPreference (OSType type, Str255 name,
                                 Handle settings);
```

`type`	The resource type of the preferences resource.
`name`	The resource name of the preferences resource.
`settings`	A handle to the data in the preferences resource.

DESCRIPTION

The `GetSoundPreference` function retrieves the block of preferences data you previously stored in a resource by calling the `SetSoundPreference` function. It is the responsibility of your component to allocate the block of data referenced by the `settings` handle. The Sound Manager resizes the handle (if necessary) and fills it with data from the resource with the specified type and name. Your sound component should dispose of the handle once it's finished reading the data from it. You can determine the size of the handle returned by the Sound Manager by calling the Memory Manager's `GetHandleSize` function.

SPECIAL CONSIDERATIONS

The `GetSoundPreference` function is available only in versions 3.0 and later of the Sound Manager.

Sound Component-Defined Routines

This section describes the routines you need to define in order to write a sound component. You need to write routines to

- load, configure, and unload your sound component
- add and remove audio sources
- read and set component settings
- control and process audio data

Some of these routines are optional for some types of sound components. All routines return result codes. If they succeed, they should return `noErr`. To simplify dispatching, the Component Manager requires these routines to return a value of type `ComponentResult`.

See "Writing a Sound Component" beginning on page 5-8 for a description of how you call these routines from within a sound component. See "Sound Manager Utilities" beginning on page 5-33 for a description of some Sound Manager utility routines you can use in a sound component.

Managing Sound Components

To write a sound component, you might need to define routines that manage the loading, configuration, and unloading of your sound component:

- `SoundComponentInitOutputDevice`
- `SoundComponentSetSource`
- `SoundComponentGetSource`
- `SoundComponentGetSourceData`
- `SoundComponentSetOutput`

After the Sound Manager opens your sound component, it attempts to add your sound component to a sound component chain. Thereafter, the Sound Manager calls your component's `SoundComponentInitOutputDevice` function to give you an opportunity to set default values for any associated hardware and to perform any hardware-specific operations.

SoundComponentInitOutputDevice

A sound output device component must implement the `SoundComponentInitOutputDevice` function. The Sound Manager calls this function to allow a sound output device component to configure any associated hardware devices.

```
pascal ComponentResult SoundComponentInitOutputDevice
                            (ComponentInstance ti, long actions);
```

`ti` A component instance that identifies your sound component.

`actions` A set of flags. This parameter is currently unused.

DESCRIPTION

Your `SoundComponentInitOutputDevice` function is called by the Sound Manager at noninterrupt time to allow your sound output device component to perform any hardware-specific initialization. You should perform any necessary initialization that

was not already performed in your `OpenComponent` function. Note that your `OpenComponent` function cannot assume that the appropriate hardware is available. As a result, the Sound Manager calls your `SoundComponentInitOutputDevice` function when it is safe to communicate with your audio hardware. You can call the `OpenMixerSoundComponent` function to create a single sound component chain.

SPECIAL CONSIDERATIONS

Your `SoundComponentInitOutputDevice` function is always called at noninterrupt time. All other component-defined routines might be called at interrupt time. Accordingly, your `SoundComponentInitOutputDevice` function should handle any remaining memory allocation needed by your component and it should lock down any relocatable blocks your component will access.

RESULT CODES

Your `SoundComponentInitOutputDevice` function should return `noErr` if successful or an appropriate result code otherwise.

SEE ALSO

See Listing 5-4 on page 5-17 for a sample `SoundComponentInitOutputDevice` function.

SoundComponentSetSource

A sound component can implement the `SoundComponentSetSource` function. The Sound Manager calls this function to identify your component's source component.

```
pascal ComponentResult SoundComponentSetSource
                            (ComponentInstance ti,
                               SoundSource sourceID,
                               ComponentInstance source);
```

`ti`	A component instance that identifies your sound component.
`sourceID`	A source ID for the source component chain created by the Apple Mixer.
`source`	A component instance that identifies your source component.

DESCRIPTION

Your `SoundComponentSetSource` function is called by the Sound Manager to identify to your sound component the sound component that is its source. The source component is identified by the `source` parameter. Your component uses that information when it

needs to obtain more data from its source (usually, by calling its `SoundComponentGetSourceData` function).

Because a sound output device component is always connected directly to one or more instances of the Apple Mixer, the `SoundComponentSetSource` function needs to be implemented only by utility components (that is, components that perform modifications on sound data). Utility components are linked together into a chain of sound components, each link of which has only one input source. As a result, a utility component can usually ignore the `sourceID` parameter passed to it.

RESULT CODES

Your `SoundComponentSetSource` function should return `noErr` if successful or an appropriate result code otherwise.

SoundComponentGetSource

A sound component can implement the `SoundComponentGetSource` function. The Sound Manager calls this function to determine your component's source component.

```
pascal ComponentResult SoundComponentGetSource
                            (ComponentInstance ti,
                               SoundSource sourceID,
                               ComponentInstance *source);
```

`ti`	A component instance that identifies your sound component.
`sourceID`	A source ID for the source component chain created by the Apple Mixer.
`source`	A component instance that identifies your source component.

DESCRIPTION

Your `SoundComponentGetSource` function is called by the Sound Manager to retrieve your component's source component instance. Your component should return, in the `source` parameter, the component instance of your component's source. This should be the source component instance your component was passed when the Sound Manager called your `SoundComponentSetSource` function.

In general, all sound components have sources, except for the source at the beginning of the source component chain. In the unlikely event that your component does not have a source, you should return `nil` in the `source` parameter. A sound output device component is always connected directly to an instance of the Apple Mixer. Accordingly, a sound output device component should return a component instance of the Apple Mixer in the `source` parameter and a source ID in the `sourceID` parameter. A utility component can ignore the `sourceID` parameter.

RESULT CODES

Your `SoundComponentGetSource` function should return `noErr` if successful or an appropriate result code otherwise.

SoundComponentGetSourceData

A utility component must implement the `SoundComponentGetSourceData` function. A sound output device component calls this function on its source component when it needs more data.

```
pascal ComponentResult SoundComponentGetSourceData
                            (ComponentInstance ti,
                               SoundComponentDataPtr *sourceData);
```

`ti` A component instance that identifies your sound component.

`sourceData`
On output, a pointer to a sound component data record that specifies the type and location of the data your component has processed.

DESCRIPTION

Your `SoundComponentGetSourceData` function is called when the sound component immediately following your sound component in the sound component chain needs more data. Your function should generate a new block of audio data, fill out a sound component data record describing the format and location of that data, and then return the address of that record in the `sourceData` parameter.

Your `SoundComponentGetSourceData` function might itself need to get more data from its source component. To do this, call through to the source component's `SoundComponentGetSourceData` function. If your component cannot generate any more data, it should set the `sampleCount` field of the sound component data record to 0 and return `noErr`.

IMPORTANT
Sound output device components do not need to implement this function, but all utility components must implement it. ▲

RESULT CODES

Your `SoundComponentGetSourceData` function should return `noErr` if successful or an appropriate result code otherwise.

SoundComponentSetOutput

A sound output device component can call the `SoundComponentSetOutput` function of the Apple Mixer to indicate the type of data it expects to receive.

```
pascal ComponentResult SoundComponentSetOutput
                            (ComponentInstance ti,
                               SoundComponentDataPtr requested,
                               SoundComponentDataPtr *actual);
```

`ti` A component instance that identifies your sound component.

`requested`
A pointer to a sound component data record that specifies the type of the data your component expects to receive.

`actual`
This parameter is currently unused.

DESCRIPTION

The Apple Mixer's `SoundComponentSetOutput` function can be called by a sound output device component to specify the kind of audio data the output device component wants to receive. The Apple Mixer uses that information to determine the type of sound component chain it needs to construct in order to deliver that kind of audio data to your sound output device component. For example, if your sound output device is able to accept 16-bit samples, the Sound Manager doesn't need to convert 16-bit audio data into 8-bit data.

The following lines of code illustrate how the sound output device component for the Apple Sound Chip might call Apple Mixer's `SoundComponentSetOutput` function:

```
myDataRec.flags = 0;                /*ignored here*/
myDataRec.format = kOffsetBinary;   /*ASC needs offset binary*/
myDataRec.sampleRate = rate22khz;   /*ASC needs 22 kHz samples*/
myDataRec.sampleSize = 8;           /*ASC needs 8-bit data*/
myDataRec.numChannels = 2;          /*ASC can do stereo*/
myDataRec.sampleCount = 1024;       /*ASC uses a 1K FIFO*/
myErr = SoundComponentSetOutput(mySource, &myDataRec, &myActual);
```

In general, however, a sound output device component shouldn't need to call the Apple Mixer's `SoundComponentSetOutput` function. Instead, it can indicate the type of data it expects to receive when it calls the `OpenMixerSoundComponent` function. The `SoundComponentSetOutput` function is intended for sophisticated sound output device components that might want to reinitialize the Apple Mixer.

IMPORTANT

Only the Apple Mixer component needs to implement this function. ▲

RESULT CODES

The Apple Mixer's `SoundComponentSetOutput` function returns `noErr` if successful or an appropriate result code otherwise.

Creating and Removing Audio Sources

To write a sound output device component, you might need to define two routines that create and remove audio sources:

- `SoundComponentAddSource`
- `SoundComponentRemoveSource`

Your component needs to contain these functions only if, like the Apple Mixer, it can mix two or more audio channels into a single output stream. Sound components that operate on a single input stream only do not need to include these functions.

SoundComponentAddSource

A sound output device component that can mix multiple channel of audio data must implement the `SoundComponentAddSource` function to add a new sound source.

```
pascal ComponentResult SoundComponentAddSource
                  (ComponentInstance ti, SoundSource *sourceID);
```

`ti`	A component instance that identifies your sound component.
`sourceID`	On exit, a source ID for the newly created source component chain.

DESCRIPTION

The `SoundComponentAddSource` function is called by the Sound Manager to create a new sound source. If your sound output device component can mix multiple channels of sound, it needs to define this function. Your `SoundComponentAddSource` function should call the Sound Manager function `OpenMixerSoundComponent` to create an new instance of the Apple Mixer component. The Apple Mixer component then creates a sound component chain capable of generating the type of data your sound output device component wants to receive.

The Apple Mixer also assigns a unique 4-byte source ID that identifies the new sound source and component chain. You can retrieve that source ID by calling the Apple Mixer's `SoundComponentAddSource` function. Your `SoundComponentAddSource` function should then pass that source ID back to the Sound Manager in the `sourceID` parameter.

IMPORTANT

Most sound components do not need to implement the `SoundComponentAddSource` function. Only sound components that can handle more than one source of input need to define it. ▲

SPECIAL CONSIDERATIONS

The `SoundComponentAddSource` function is called at noninterrupt time.

RESULT CODES

Your `SoundComponentAddSource` function should return `noErr` if successful or an appropriate result code otherwise.

SEE ALSO

See page 5-33 for a description of `OpenMixerSoundComponent`.

SoundComponentRemoveSource

A sound output device component that implements the `SoundComponentAddSource` function must also implement the `SoundComponentRemoveSource` function to remove sound sources.

```
pascal ComponentResult SoundComponentRemoveSource
                    (ComponentInstance ti, SoundSource sourceID);
```

`ti`	A component instance that identifies your sound component.
`sourceID`	A source ID for the source component chain to be removed.

DESCRIPTION

Your `SoundComponentRemoveSource` function is called by the Sound Manager to remove the existing sound source specified by the `sourceID` parameter. Your `SoundComponentRemoveSource` function should do whatever is necessary to invalidate that source and then call through to the Apple Mixer's `SoundComponentRemoveSource` function.

IMPORTANT

Most sound components do not need to implement the `SoundComponentRemoveSource` function. Only sound components that can handle more than one source of input need to define it. ▲

SPECIAL CONSIDERATIONS

Your `SoundComponentRemoveSource` function is always called at noninterrupt time.

RESULT CODES

Your `SoundComponentRemoveSource` function should return `noErr` if successful or an appropriate result code otherwise.

Getting and Setting Sound Component Information

To write a sound component, you need to define two routines that determine the capabilities of your component or to change those capabilities:

- `SoundComponentGetInfo`
- `SoundComponentSetInfo`

SoundComponentGetInfo

A sound component must implement the `SoundComponentGetInfo` function. The Sound Manager calls this function to get information about the capabilities of your component.

```
pascal ComponentResult SoundComponentGetInfo
                            (ComponentInstance ti,
                               SoundSource sourceID,
                               OSType selector, void *infoPtr);
```

`ti`	A component instance that identifies your sound component.
`sourceID`	A source ID for a source component chain.
`selector`	A sound component information selector. See "Sound Component Information Selectors" beginning on page 5-22 for a description of the available selectors.
`infoPtr`	On output, a pointer to the information requested by the caller.

DESCRIPTION

Your `SoundComponentGetInfo` function returns information about your sound component. The `sourceID` parameter specifies the sound source to return information about, and the `selector` parameter specifies the kind of information to be returned. If the information occupies 4 or fewer bytes, it should be returned in the location pointed to by the `infoPtr` parameter. If the information is larger than 4 bytes, the `infoPtr` parameter is a pointer to a component information list, a 6-byte structure of type `SoundInfoList`:

```
typedef struct {
   short          count;
   Handle         handle;
} SoundInfoList, *SoundInfoListPtr;
```

This structure consists of a count and a handle to a variable-sized array. The `count` field specifies the number of elements in the array to which `handle` is a handle. It is your component's responsibility to allocate the block of data referenced by that handle, but it is the caller's responsibility to dispose of that handle once it is finished with it.

The data type of the array elements depends on the kind of information being returned. For example, the selector `siSampleSizeAvailable` indicates that you should return a list of the sample sizes your component can support. You return the information by passing back, in the `infoPtr` parameter, a pointer to an integer followed by a handle to an array of integers.

If your component cannot provide the information specified by the `selector` parameter, it should pass the selector to its source component.

SPECIAL CONSIDERATIONS

Your `SoundComponentGetInfo` function is not called at interrupt time if it is passed a selector that might cause it to allocate memory for the handle in the component information list.

RESULT CODES

Your `SoundComponentGetInfo` function should return `noErr` if successful or an appropriate result code otherwise.

SEE ALSO

See "Finding and Changing Component Capabilities" on page 5-18 for a sample `SoundComponentGetInfo` function.

SoundComponentSetInfo

A sound component must implement the `SoundComponentSetInfo` function. The Sound Manager calls this function to modify settings of your component.

```
pascal ComponentResult SoundComponentSetInfo
                           (ComponentInstance ti,
                              SoundSource sourceID,
                              OSType selector, void *infoPtr);
```

`ti` A component instance that identifies your sound component.

`sourceID`	A source ID for a source component chain.
`selector`	A sound component information selector. See "Sound Component Information Selectors" beginning on page 5-22 for a description of the available selectors.
`infoPtr`	A pointer to the information your component is to use to modify its settings. If the information occupies 4 or fewer bytes, however, this parameter contains the information itself, not the address of the information.

DESCRIPTION

Your `SoundComponentSetInfo` function is called by the Sound Manager to set one of the settings for your component, as specified by the `selector` parameter. If the information associated with that selector occupies 4 or fewer bytes, it is passed on the stack, in the `infoPtr` parameter itself. Otherwise, the `infoPtr` parameter is a pointer to a structure of type `SoundInfoList`. See the description of `SoundComponentGetInfo` for more information about the `SoundInfoList` structure.

If your component cannot modify the settings specified by the `selector` parameter, it should pass the selector to its source component.

RESULT CODES

Your `SoundComponentSetInfo` function should return `noErr` if successful or an appropriate result code otherwise.

Managing Source Data

To write a sound output device component, you might need to define routines that manage the flow of data in a sound channel:

- `SoundComponentStartSource`
- `SoundComponentStopSource`
- `SoundComponentPauseSource`
- `SoundComponentPlaySourceBuffer`

SoundComponentStartSource

A sound output device component must implement the `SoundComponentStartSource` function. The Sound Manager calls this function to start playing sounds in one or more sound channels.

```
pascal ComponentResult SoundComponentStartSource
                            (ComponentInstance ti,
                               short count, SoundSource *sources);
```

`ti`	A component instance that identifies your sound component.
`count`	The number of source IDs in the array pointed to by the `source` parameter.
`sources`	An array of source IDs.

DESCRIPTION

Your `SoundComponentStartSource` function is called by the Sound Manager to begin playing the sounds originating from the sound sources specified by the `sources` parameter. Your function should start (or resume) sending data from those sources to the associated sound output device. If your component supports only one sound source, you can ignore the `sources` parameter.

SPECIAL CONSIDERATIONS

Your `SoundComponentStartSource` function can be called at interrupt time.

RESULT CODES

Your `SoundComponentStartSource` function should return `noErr` if successful or an appropriate result code otherwise. You should return `noErr` even if no sounds are playing in the specified channels.

SoundComponentStopSource

A sound output device component must implement the `SoundComponentStopSource` function. The Sound Manager calls this function to stop playing sounds in one or more sound channels.

```
pascal ComponentResult SoundComponentStopSource
                            (ComponentInstance ti, short count,
                               SoundSource *sources);
```

`ti`	A component instance that identifies your sound component.
`count`	The number of source IDs in the array pointed to by the `source` parameter.
`sources`	An array of source IDs.

DESCRIPTION

Your `SoundComponentStopSource` function is called by the Sound Manager to stop the sounds originating from the sound sources specified by the `sources` parameter. Your function should stop sending data from those sources to the associated sound output device. In addition, your `SoundComponentStopSource` function should flush any data from the specified sound sources that it's caching. If your component supports only one sound source, you can ignore the `sources` parameter.

RESULT CODES

Your `SoundComponentStopSource` function should return `noErr` if successful or an appropriate result code otherwise. You should return `noErr` even if no sounds are playing in the specified channels.

SoundComponentPauseSource

A sound output device component must implement the `SoundComponentPauseSource` function. The Sound Manager calls this function to stop pause the playing of sounds in one or more sound channels.

```
pascal ComponentResult SoundComponentPauseSource
                            (ComponentInstance ti,
                               short count, SoundSource *sources);
```

`ti`	A component instance that identifies your sound component.
`count`	The number of source IDs in the array pointed to by the `source` parameter.
`sources`	An array of source IDs.

DESCRIPTION

Your `SoundComponentPauseSource` function is called by the Sound Manager to pause the playing of the sounds originating from the sound sources specified by the `sources` parameter. Your function should stop sending data from those sources to the associated sound output device. Because your `SoundComponentStartSource` function might be called to resume playing sounds, you should not flush any data. If your component supports only one sound source, you can ignore the `sources` parameter.

RESULT CODES

Your `SoundComponentPauseSource` function should return `noErr` if successful or an appropriate result code otherwise. You should return `noErr` even if no sounds are playing in the specified channels.

SoundComponentPlaySourceBuffer

A sound component must implement the `SoundComponentPlaySourceBuffer` function. The Sound Manager calls this function to start a new sound playing.

```
pascal ComponentResult SoundComponentPlaySourceBuffer
                                (ComponentInstance ti,
                                   SoundSource sourceID,
                                   SoundParamBlockPtr pb,
                                   long actions);
```

`ti`	A component instance that identifies your sound component.
`sourceID`	A source ID for a source component chain.
`pb`	A pointer to a sound parameter block.
`actions`	A set of 32 bit flags that describe the actions to be taken when preparing to play the source data. See "Action Flags" on page 5-27 for a description of the constants you can use to select bits in this parameter.

DESCRIPTION

Your `SoundComponentPlaySourceBuffer` function is called by the Sound Manager to start a new sound playing. The sound parameter block pointed to by the `pb` parameter specifies the sound to be played. That parameter block should be passed successively to all sound components in the chain specified by the `sourceID` parameter. This allows the components to determine their output formats and playback settings and to prepare for a subsequent call to their `SoundComponentGetSourceData` function. It also allows a sound output device component to prepare for starting up its associated hardware.

RESULT CODES

Your `SoundComponentPlaySourceBuffer` function should return `noErr` if successful or an appropriate result code otherwise.

Summary of Sound Components

This section provides a C summary for the constants, data types, and routines you can use to write a sound component. There are currently no Pascal interfaces available for writing sound components.

C Summary

Constants

```
/*component types*/
#define kSoundComponentType      'sift'   /*utility component*/
#define kMixerType               'mixr'   /*mixer component*/
#define kSoundHardwareType       'sdev'   /*sound output device component*/
#define kSoundCompressor         'scom'   /*compression component*/
#define kSoundDecompressor       'sdec'   /*decompression component*/
#define kNoSoundComponentType    '****'   /*no type*/

/*subtypes for kSoundComponentType component type*/
#define kRate8SubType            'ratb'   /*8-bit rate converter*/
#define kRate16SubType           'ratw'   /*16-bit rate converter*/
#define kConverterSubType        'conv'   /*sample format converter*/
#define kSndSourceSubType        'sour'   /*generic source component*/

/*subtypes for kMixerType component type*/
#define kMixer8SubType           'mixb'   /*8-bit mixer*/
#define kMixer16SubType          'mixw'   /*16-bit mixer*/

/*subtypes for kSoundHardwareType component type*/
#define kClassicSubType          'clas'   /*Classic hardware*/
#define kASCSubType              'asc '   /*ASC device*/
#define kDSPSubType              'dsp '   /*DSP device*/

/*subtypes for kSoundCompressor and kSoundDecompressor component types*/
#define kMace3SubType            'MAC3'   /*MACE 3:1*/
#define kMace6SubType            'MAC6 '  /*MACE 6:1*/
#define kCDXA4SubType            'CDX4'   /*CD/XA 4:1*/
#define kCDXA2SubType            'CDX2'   /*CD/XA 2:1*/

#define kSoundComponentCodeType  'sift'   /*sound component code type*/
```

```
/*first selector that can be delegated up the chain*/
#define kDelegatedSoundComponentSelectors       0x0100

/*Component Manager selectors for routines*/
enum {
   /*the following calls cannot be delegated*/
   kSoundComponentInitOutputDeviceSelect     = 1,
   kSoundComponentSetSourceSelect,
   kSoundComponentGetSourceSelect,
   kSoundComponentGetSourceDataSelect,
   kSoundComponentSetOutputSelect,
   /*the following calls can be delegated*/
   kSoundComponentAddSourceSelect = kDelegatedSoundComponentSelectors + 1,
   kSoundComponentRemoveSourceSelect,
   kSoundComponentGetInfoSelect,
   kSoundComponentSetInfoSelect,
   kSoundComponentStartSourceSelect,
   kSoundComponentStopSourceSelect,
   kSoundComponentPauseSourceSelect,
   kSoundComponentPlaySourceBufferSelect
};

/*sound component information selectors*/
#define siChannelAvailable       'chav'   /*number of channels available*/
#define siCompressionAvailable   'cmav'   /*compression types available*/
#define siCompressionFactor      'cmfa'   /*current compression factor*/
#define siCompressionType        'comp'   /*current compression type*/
#define siHardwareMute           'hmut'   /*current hardware mute state*/
#define siHardwareVolume         'hvol'   /*current hardware volume*/
#define siHardwareVolumeSteps    'hstp'   /*number of hardware volume steps*/
#define siHeadphoneMute          'pmut'   /*current headphone mute state*/
#define siHeadphoneVolume        'pvol'   /*current headphone volume*/
#define siHeadphoneVolumeSteps   'hdst'   /*num. of headphone volume steps*/
#define siNumberChannels         'chan'   /*current number of channels*/
#define siQuality                'qual'   /*current quality*/
#define siRateMultiplier         'rmul'   /*current rate multiplier*/
#define siSampleRate             'srat'   /*current sample rate*/
#define siSampleRateAvailable    'srav'   /*sample rates available*/
#define siSampleSize             'ssiz'   /*current sample size*/
#define siSampleSizeAvailable    'ssav'   /*sample sizes available*/
#define siSpeakerMute            'smut'   /*current speaker mute*/
#define siSpeakerVolume          'svol'   /*current speaker volume*/
#define siVolume                 'volu'   /*current volume setting*/
```

```
/*audio data format types*/
#define kOffsetBinary            'raw '
#define kTwosComplement          'twos'
#define kMACE3Compression        'MAC3'
#define kMACE6Compression        'MAC6'

/*sound component features flags*/
#define k8BitRawIn               (1 << 0)    /*data flags*/
#define k8BitTwosIn              (1 << 1)
#define k16BitIn                 (1 << 2)
#define kStereoIn                (1 << 3)
#define k8BitRawOut              (1 << 8)
#define k8BitTwosOut             (1 << 9)
#define k16BitOut                (1 << 10)
#define kStereoOut               (1 << 11)

#define kReverse                 (1 << 16)   /*action flags*/
#define kRateConvert             (1 << 17)
#define kCreateSoundSource       (1 << 18)

#define kHighQuality             (1 << 22)   /*performance flags*/
#define kRealTime                (1 << 23)

/*action flags for SoundComponentPlaySourceBuffer*/
#define kSourcePaused            (1 << 0)
#define kPassThrough             (1 << 16)
#define kNoSoundComponentChain   (1 << 17)

/*flags for OpenMixerSoundComponent*/
#define kNoMixing                (1 << 0)    /*don't mix sources*/
#define kNoSampleRateConversion  (1 << 1)    /*don't convert sample rate*/
#define kNoSampleSizeConversion  (1 << 2)    /*don't convert sample size*/
#define kNoSampleFormatConversion            \
                                 (1 << 3)    /*don't convert sample format*/
#define kNoChannelConversion     (1 << 4)    /*don't convert stereo/mono*/
#define kNoDecompression         (1 << 5)    /*don't decompress*/
#define kNoVolumeConversion      (1 << 6)    /*don't apply volume*/
#define kNoRealtimeProcessing    (1 << 7)    /*don't run at interrupt time*/

/*quality flags*/
#define kBestQuality             (1 << 0)    /*use interp. in rate conv.*/
```

```
/*volume specifications*/
#define kSilenceByte            0x80
#define kSilenceLong            0x80808080
#define kFullVolume             0x0100
```

Data Types

Unsigned Fixed-Point Numbers

```
typedef unsigned long UnsignedFixed;   /*unsigned fixed-point number*/
```

Sound Component Data Record

```
typedef struct {
   long           flags;              /*sound component flags*/
   OSType         format;             /*data format*/
   short          numChannels;        /*number of channels in data*/
   short          sampleSize;         /*size of a sample*/
   UnsignedFixed  sampleRate;         /*sample rate*/
   long           sampleCount;        /*number of samples in buffer*/
   Byte           *buffer;            /*location of data*/
   long           reserved;           /*reserved*/
} SoundComponentData, *SoundComponentDataPtr;
```

Sound Parameter Block

```
typedef pascal Boolean (*SoundParamProcPtr)(SoundParamBlockPtr *pb);

struct SoundParamBlock {
   long                recordSize;    /*size of this record in bytes*/
   SoundComponentData  desc;          /*description of sound buffer*/
   Fixed               rateMultiplier;/*rate multiplier*/
   short               leftVolume;    /*volume on left channel*/
   short               rightVolume;   /*volume on right channel*/
   long                quality;       /*quality*/
   ComponentInstance   filter;        /*filter*/
   SoundParamProcPtr   moreRtn;       /*routine to call to get more data*/
   SoundParamProcPtr   completionRtn; /*buffer complete routine*/
   long                refCon;        /*user refcon*/
   short               result;        /*result*/
};
typedef struct SoundParamBlock SoundParamBlock;
typedef SoundParamBlock *SoundParamBlockPtr;
```

Sound Source

```
typedef struct privateSoundSource *SoundSource;
```

Sound Information List

```
typedef struct {
   short          count;
   Handle         handle;
} SoundInfoList, *SoundInfoListPtr;
```

Compression Information Record

```
typedef struct {
   long           recordSize;
   OSType         format;
   short          compressionID;
   short          samplesPerPacket;
   short          bytesPerPacket;
   short          bytesPerFrame;
   short          bytesPerSample;
   short          futureUse1;
} CompressionInfo, *CompressionInfoPtr, **CompressionInfoHandle;
```

Sound Manager Utilities

Opening and Closing the Apple Mixer Component

```
pascal OSErr OpenMixerSoundComponent
                           (SoundComponentDataPtr outputDescription,
                            long outputFlags,
                            ComponentInstance *mixerComponent);
pascal OSErr CloseMixerSoundComponent
                           (ComponentInstance ci);
```

Saving and Restoring Sound Component Preferences

```
pascal OSErr SetSoundPreference
                           (OSType type, Str255 name, Handle settings);
pascal OSErr GetSoundPreference
                           (OSType type, Str255 name, Handle settings);
```

Sound Component-Defined Routines

Managing Sound Components

```
pascal ComponentResult SoundComponentInitOutputDevice
                            (ComponentInstance ti, long actions);
pascal ComponentResult SoundComponentSetSource
                            (ComponentInstance ti, SoundSource sourceID,
                             ComponentInstance source);
pascal ComponentResult SoundComponentGetSource
                            (ComponentInstance ti, SoundSource sourceID,
                             ComponentInstance *source);
pascal ComponentResult SoundComponentGetSourceData
                            (ComponentInstance ti,
                             SoundComponentDataPtr *sourceData);
pascal ComponentResult SoundComponentSetOutput
                            (ComponentInstance ti,
                             SoundComponentDataPtr requested,
                             SoundComponentDataPtr *actual);
```

Creating and Removing Audio Sources

```
pascal ComponentResult SoundComponentAddSource
                            (ComponentInstance ti, SoundSource *sourceID);
pascal ComponentResult SoundComponentRemoveSource
                            (ComponentInstance ti, SoundSource sourceID);
```

Getting and Setting Sound Component Information

```
pascal ComponentResult SoundComponentGetInfo
                            (ComponentInstance ti, SoundSource sourceID,
                             OSType selector, void *infoPtr);
pascal ComponentResult SoundComponentSetInfo
                            (ComponentInstance ti, SoundSource sourceID,
                             OSType selector, void *infoPtr);
```

Managing Source Data

```
pascal ComponentResult SoundComponentStartSource
                            (ComponentInstance ti, short count,
                             SoundSource *sources);
pascal ComponentResult SoundComponentStopSource
                            (ComponentInstance ti, short count,
                             SoundSource *sources);
```

```
pascal ComponentResult SoundComponentPauseSource
                            (ComponentInstance ti, short count,
                             SoundSource *sources);
pascal ComponentResult SoundComponentPlaySourceBuffer
                            (ComponentInstance ti, SoundSource sourceID,
                             SoundParamBlockPtr pb, long actions);
```

Assembly-Language Summary

Data Structures

Sound Component Data Record

0	`flags`	long	sound component flags
4	`format`	long	data format
8	`numChannels`	word	number of channels in data
10	`sampleSize`	word	size of a sample
12	`sampleRate`	long	sample rate (Fixed)
16	`sampleCount`	long	number of samples in buffer
20	`buffer`	long	location of data
24	`reserved`	long	reserved

Sound Parameter Block

0	`recordSize`	long	size of this record in bytes
4	`desc`	28 bytes	description of sound buffer
32	`rateMultiplier`	long	rate multiplier (Fixed)
36	`leftVolume`	word	volume on left channel
38	`rightVolume`	word	volume on right channel
40	`quality`	long	quality
44	`filter`	long	filter
48	`moreRtn`	long	routine to call to get more data
52	`completionRtn`	long	buffer complete routine
56	`refCon`	long	user refcon
60	`result`	word	result

Sound Information List

0	`count`	word	number of data items in the handle
2	`handle`	long	handle to list of data items

Compression Information Record

0	`recordSize`	long	the size of this record
4	`format`	4 bytes	compression format
8	`compressionID`	word	compression ID
10	`samplesPerPacket`	word	the number of samples per packet
12	`bytesPerPacket`	word	the number of bytes per packet
14	`bytesPerFrame`	word	the number of bytes per frame
16	`bytesPerSample`	word	the number of bytes per sample
18	`futureUse1`	word	reserved