Audio Components

This chapter describes audio components, which are code modules used by the Sound Manager to adjust volumes or other settings of a sound output device. In general, you need to write an audio component only if you are developing a sound output device with multiple output ports that can be independently controlled by software. If your sound output device has only one software-controllable output port, the sound output device component for that device manages the volume levels of the port.

IMPORTANT
The Sound Manager loads and manages audio components, which operate transparently to applications. The routines described in this chapter are intended for use exclusively by audio components. ▲

To use this chapter, you should already be familiar with writing sound output device components, as described in the chapter "Sound Components" in this book. Because audio components are components, you also need to be familiar with the Component Manager, described in *Inside Macintosh: More Macintosh Toolbox*.

This chapter begins by describing what audio components are and the Sound Manager uses them. Then it provides instructions on how to write an audio component. The section "Audio Components Reference" beginning on page 6-8 describes the routines that your audio component might need to define.

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

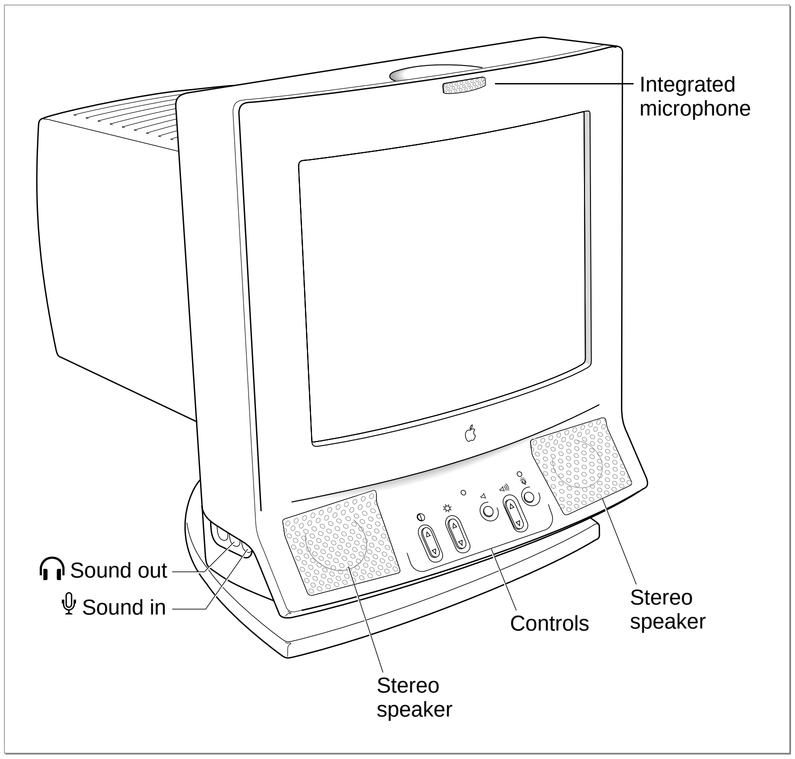
About Audio Components

An **audio component** is a component that works with the Sound Manager to adjust volumes or other settings of a sound output device. The Sound Manager uses audio components, however, only when a particular sound output device has more than one audio port that can be controlled through software. If a sound output device has only one audio port, the sound component that communicates with the output device controls the volume settings of that port.

IMPORTANT
Because audio components are currently used to manage only volume and mute settings, they might have been called *volume components*. The more general term anticipates future capabilities of audio components. For example, audio components might in the future be used to modify bass or treble settings of an audio port. ▲

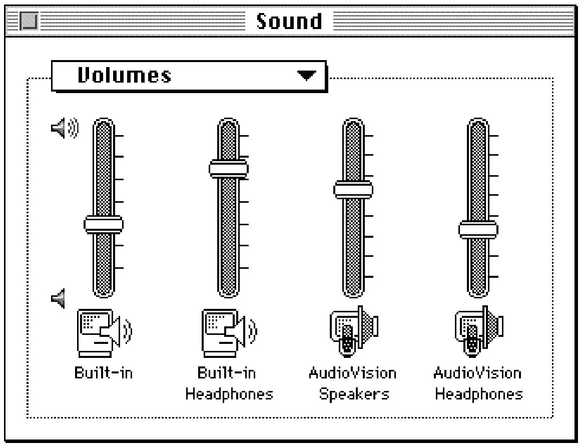
An **audio port** is any independently controllable sound-producing hardware connected or attached to a sound output device. For example, the Apple AudioVision 14 Display (shown in Figure 6-1) contains two audio ports: a set of speakers and a jack for headphones.

Figure 6-1 The Apple AudioVision 14 Display



As the Volumes subpanel of the Sound control panel shows (Figure 6-2), the two audio ports are independently controllable by software.

Figure 6-2 The Volumes control panel for the Apple AudioVision 14 Display



The control panel shown in Figure 6-2 contains volume sliders both for the set of speakers and for the headphones. The volume of the speakers is controlled by the sound component that drives the sound output device. The volume of the headphones is controlled by an audio component.

In short, audio components are used to allow a single sound output device to have more than one audio port. The sound component that communicates with that device can control the volume setting of one audio port; audio components control the volume settings of all other audio ports.

Writing an Audio Component

Because an audio component is a component, it must be able to respond to standard selectors sent by the Component Manager. In addition, an audio component must handle other selectors specific to audio components. This section briefly describes how to write an audio component.

Creating an Audio Component

An audio 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, an audio component must contain code to handle required selectors passed to it by the Component Manager as well as selectors specific to the audio 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 audio 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 `kAudioCodeType`.

```
#define kAudioCodeType     'adio'   /*audio 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 `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 audio components, the `componentType` field must be set to a value recognized by the Sound Manager.

```
#define kAudioComponentType      'adio'   /*audio 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 audio components have these subtypes:

```
#define kAwacsPhoneSubType             'hphn'   /*AWACS phone*/
#define kAudioVisionSpeakerSubType     'telc'   /*AudioVision speaker*/
#define kAudioVisionHeadphoneSubType   'telh'   /*AudioVision headphones*/
```

If you write an audio 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 audio component in this field.

The `componentFlags` field of the component description for an audio 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 useful for audio 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.

You should set the `componentFlagsMask` field to 0.

Your audio 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 audio component contains a `'BNDL'` resource, then the file's bundle bit must be set.

Dispatching to Audio Component-Defined Routines

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

```
pascal ComponentResult MyAudioDispatch (ComponentParameters *params,
                                        AudioGlobalsPtr globals);
```

The Component Manager calls your sound component by passing `MyAudioDispatch` a selector in the `params->what` field; `MyAudioDispatch` must interpret the selector and possibly dispatch to some other routine in the resource. Your audio 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 audio component must be able to respond to component-specific selectors. The Sound Manager can pass these selectors to your audio component:

```
enum {
   kAudioGetVolumeSelect = 0,
   kAudioSetVolumeSelect,
   kAudioGetMuteSelect,
   kAudioSetMuteSelect,
   kAudioSetToDefaultsSelect,
   kAudioGetInfoSelect
};
```

You can respond to these selectors by calling the Component Manager routine `CallComponentFunctionWithStorage`. See the section "Audio Component-Defined Routines" beginning on page 6-9 for information on how to handle these selectors.

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 audio 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 audio component is closed. Their maps will be left in the subheap when the subheap is freed, causing the Resource Manager to crash. ▲

Audio Components Reference

This section describes the data structures you can use to write an audio component. It also describes the routines that your audio component should call in response to an audio component selector. See "Writing an Audio Component" beginning on page 6-5 for information on creating a component that contains these component-defined routines.

Data Structures

This section describes the data structure you need to use when writing an audio component.

Audio Information Records

You return information about the capabilities of your audio component in the `info` parameter passed to your `AudioGetInfo` function. The `info` parameter contains a pointer to an **audio information record.** An audio information record is defined by the `AudioInfo` data type.

```
typedef struct {
   long           capabilitiesFlags;   /*device capabilities*/
   long           reserved;            /*reserved*/
   unsigned short numVolumeSteps;      /*number of volume steps*/
} AudioInfo, *AudioInfoPtr;
```

Field descriptions

`capabilitiesFlags`
A set of bit flags specifying the capabilities of the audio component. You can use constants to set some of these bits:

```
#define audioDoesMono                  (1L<<0)  /*supports mono output*/
#define audioDoesStereo                (1L<<1)  /*supports stereo output*/
#define audioDoesIndependentChannels   (1L<<2)  /*supports independent
                                                   software control of each channel*/
```

`reserved` Reserved for use by Apple Computer, Inc.

`numVolumeSteps`
The number of volume steps your audio component supports.

Audio Component-Defined Routines

This section describes the routines you must define in order to write an audio component. You need to write routines to

- get and set volume levels of a sound output device
- manage mute states
- reset device settings
- get information about the audio component

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 an Audio Component" beginning on page 6-5 for a description of how you call these routines from within an audio component.

Getting and Setting Volumes

To write an audio component, you might need to define two routines that manage the volume level of the associated audio port:

- `AudioGetVolume`
- `AudioSetVolume`

AudioGetVolume

An audio component can implement the `AudioGetVolume` function. The Sound Manager calls this function to determine the current volume of an audio port.

```
pascal ComponentResult AudioGetVolume (ComponentInstance ac,
                                          short whichChannel,
                                          ShortFixed *volume);
```

`ac`	A component instance that identifies your audio component.
`whichChannel`	The channel or channels whose volume you should return.
`volume`	On output, the current volume level of the specified channel.

DESCRIPTION

Your `AudioGetVolume` function is called by the Sound Manager to determine the current volume levels of one or more channels of an audio port. The `volume` parameter can have any value between 0 and 1, where 0 indicates minimum volume and 1 indicates maximum volume. The `whichChannel` parameter indicates the channels or channels whose volumes you should return. The following constants are defined for the `whichChannel` parameter:

```
#define audioAllChannels        0       /*all channels*/
#define audioLeftChannel        1       /*left channel*/
#define audioRightChannel       2       /*right channel*/
```

RESULT CODES

Your `AudioGetVolume` function should return `noErr` if successful or an appropriate result code otherwise. In particular, if your audio component doesn't support software control of volume levels, `AudioGetVolume` should return `unImpErr`.

AudioSetVolume

An audio component can implement the `AudioSetVolume` function. The Sound Manager calls this function to set the current volume of an audio port.

```
pascal ComponentResult AudioSetVolume (ComponentInstance ac,
                                          short whichChannel,
                                          ShortFixed volume);
```

`ac`	A component instance that identifies your audio component.
`whichChannel`	The channel or channels whose volume you should set.
`volume`	The desired volume level of the specified channel.

DESCRIPTION

Your `AudioSetVolume` function is called by the Sound Manager to set the volume levels of one or more channels of an audio port. See the description of the `AudioGetVolume` function for the values of the `whichChannel` and `volume` parameters.

RESULT CODES

Your `AudioSetVolume` function should return `noErr` if successful or an appropriate result code otherwise. In particular, if your audio component doesn't support software control of volume levels, `AudioSetVolume` should return `unImpErr`.

Managing the Mute State

To write an audio component, you might need to define two routines that manage the mute state of the associated audio port:

- `AudioGetMute`
- `AudioSetMute`

AudioGetMute

An audio component can implement the `AudioGetMute` function. The Sound Manager calls this function to determine the current mute state of an audio port.

```
pascal ComponentResult AudioGetMute (ComponentInstance ac,
                                        short whichChannel,
                                        short *mute);
```

`ac` A component instance that identifies your audio component.

`whichChannel`
The channel or channels whose mute state you should return.

`mute`
On output, the current mute state of the specified channel.

DESCRIPTION

Your `AudioGetMute` function is called by the Sound Manager to determine the current mute state of one or more channels of an audio port. The following constants define the mute states you can return in the `mute` parameter:

```
#define audioUnmuted            0       /*device is not muted*/
#define audioMuted              1       /*device is muted*/
```

The `whichChannel` parameter indicates the channels or channels whose mute state you should return. The following constants are defined for the `whichChannel` parameter:

```
#define audioAllChannels        0       /*all channels*/
#define audioLeftChannel        1       /*left channel*/
#define audioRightChannel       2       /*right channel*/
```

RESULT CODES

Your `AudioGetMute` function should return `noErr` if successful or an appropriate result code otherwise. In particular, if your audio component doesn't support software control of mute states, `AudioGetMute` should return `unImpErr`.

AudioSetMute

An audio component can implement the `AudioSetMute` function. The Sound Manager calls this function to set the current mute state of an audio port.

```
pascal ComponentResult AudioSetMute (ComponentInstance ac,
                                       short whichChannel,
                                       short mute);
```

`ac` A component instance that identifies your audio component.

`whichChannel`
The channel or channels whose mute state you should set.

`mute`
The desired mute state of the specified channel.

DESCRIPTION

Your `AudioSetMute` function is called by the Sound Manager to set the mute state of one or more channels of an audio port. See the description of the `AudioGetMute` function for the values of the `whichChannel` and `mute` parameters.

RESULT CODES

Your `AudioSetMute` function should return `noErr` if successful or an appropriate result code otherwise. In particular, if your audio component doesn't support software control of mute states, `AudioSetMute` should return `unImpErr`.

Resetting Audio Components

To write an audio component, you need to define the `AudioSetToDefaults` routine, which resets the associated audio port to its default settings.

AudioSetToDefaults

The Sound Manager might call your `AudioSetToDefaults` function to reset an audio port.

```
pascal ComponentResult AudioSetToDefaults (ComponentInstance ac);
```

`ac`	A component instance that identifies your audio component.

DESCRIPTION

Your `AudioSetToDefaults` function should reset its volume and mute levels to some reasonable default value. It should also reset to reasonable values any other settings it might be maintaining privately.

RESULT CODES

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

Getting Audio Component Information

To write an audio component, you need to define the `AudioGetInfo` routine, which returns information about the capabilities of your component.

AudioGetInfo

An audio component must implement the `AudioGetInfo` function. The Sound Manager calls this function to get information about the capabilities of your component.

```
pascal ComponentResult AudioGetInfo (ComponentInstance ac,
                                                  AudioInfoPtr info);
```

`ac`	A component instance that identifies your sound component.
`info`	A pointer to an audio information record.

DESCRIPTION

Your `AudioGetInfo` function returns information about your audio component. You should fill out the audio information record pointed to by the `info` parameter. See "Audio Information Records" beginning on page 6-9 for a description of the audio information record.

RESULT CODES

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

Summary of Audio Components

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

C Summary

Constants

```
/*component types*/
#define kAudioComponentType            'adio'   /*audio component*/

/*subtypes for kAudioComponentType component type*/
#define kAwacsPhoneSubType             'hphn'   /*AWACS phone*/
#define kAudioVisionSpeakerSubType     'telc'   /*AudioVision speaker*/
#define kAudioVisionHeadphoneSubType   'telh'   /*AudioVision headphones*/

#define kAudioCodeType                 'adio'   /*audio component code type*/

/*Component Manager selectors for routines*/
enum {
   kAudioGetVolumeSelect = 0,
   kAudioSetVolumeSelect,
   kAudioGetMuteSelect,
   kAudioSetMuteSelect,
   kAudioSetToDefaultsSelect,
   kAudioGetInfoSelect
};

/*values for whichChannel parameter*/
#define audioAllChannels         0        /*all channels*/
#define audioLeftChannel         1        /*left channel*/
#define audioRightChannel        2        /*right channel*/

/*values for mute parameter*/
#define audioUnmuted             0        /*device is not muted*/
#define audioMuted               1        /*device is muted*/
```

```
/*audio component features flags*/
#define audioDoesMono                   (1L<<0)  /*supports mono output*/
#define audioDoesStereo                 (1L<<1)  /*supports stereo output*/
#define audioDoesIndependentChannels    (1L<<2)  /*supports independent
                                           software control of each channel*/
```

Data Types

Short Fixed-Point Numbers

```
typedef short ShortFixed;
```

Audio Information Record

```
typedef struct {
   long           capabilitiesFlags;   /*device capabilities*/
   long           reserved;            /*reserved*/
   unsigned short numVolumeSteps;      /*number of volume steps*/
} AudioInfo, *AudioInfoPtr;
```

Audio Component-Defined Routines

Getting and Setting Volumes

```
pascal ComponentResult AudioGetVolume
                            (ComponentInstance ac, short whichChannel,
                             ShortFixed *volume);
pascal ComponentResult AudioSetVolume
                            (ComponentInstance ac, short whichChannel,
                             ShortFixed volume);
```

Managing the Mute State

```
pascal ComponentResult AudioGetMute
                            (ComponentInstance ac, short whichChannel,
                             short *mute);
pascal ComponentResult AudioSetMute
                            (ComponentInstance ac, short whichChannel,
                             short mute);
```

Resetting Audio Components

```
pascal ComponentResult AudioSetToDefaults
                            (ComponentInstance ac);
```

Getting Audio Component Information

```
pascal ComponentResult AudioGetInfo
                            (ComponentInstance ac, AudioInfoPtr info);
```

Assembly-Language Summary

Data Structures

Audio Information Record

0	`capabilitiesFlags`	long	device capabilities
4	`reserved`	long	reserved
8	`numVolumeSteps`	word	number of volume steps