Developer Note

AV Architecture

Developer Press

Apple Computer, Inc.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures and Tables

About This Note

This developer note describes the Apple AV Architecture, which is an essential element in Apple Computer's upcoming audio and video products. The note provides the information needed by developers to understand the functions of the architecture, and to take advantage of these features to develop components that fit into the architecture.

The note assumes that you are familiar with the functionality and programming requirements of Apple Macintosh computers. It is intended to be used in conjunction with the developer notes published for different Apple devices that implement the AV Architecture. It also assumes that you have read the "Component Manager" chapter of *Inside Macintosh: More Macintosh Toolbox*, since the AV Architecture relies heavily on the Component Manager.

The note consists of six chapters, an appendix, a glossary, and an index.

- Chapter 1, "Overview of the AV Architecture and Application Software," introduces the concept of the new architecture, explains why the architecture was needed, and gives an overview of the software components that make up the architecture.
- Chapter 2, "Panel Components," describes the sequence of calls to each of the Panel Component functions. It explains how the panel owner searches for candidate panels, opens panels for display to the user, runs the setup application, and then closes the panel window.
- Chapter 3, "Engine Components," explains why Engine Components are critical to the AV Architecture, and describes the calls to each of the Engine Component functions. It also describes how to register Engine Components multiple times, and explains how the Engine Components notify all other components when there are changes in the machine state.
- Chapter 4, "Port Components," explains why ports and Port Components are critical to the AV Architecture. It also describes the calls to each of the Port Component functions.
- Chapter 5, "Device Components," explains why Device Components are critical to the AV Architecture. It also describes the calls to each of the Device Component functions.
- Chapter 6, "More Information about the AV Architecture," provides supplementary information, including how to store data in components, how to manage the storage, how the notification mechanism works, what utility components are available, and an overview of the library of utility functions.
- The appendix, "Manager Component," provides information about a component that may be used with the AV Architecture, but is not a required part of the architecture. The Manager Component can load

other components; register Device and Port Components with the Display Manager; establish connections between Device and Port Components; and register or unregister components when the system is initialized, shuts down, goes into sleep mode, or wakes up from sleep mode.

Conventions Used in This Note

The following conventions are used throughout this note

Note
This type of note contains information of general interest. ◆

IMPORTANT
A note like this contains important information that you should read before proceeding. ▲

Terms in **boldface** type are defined in the glossary.

A special font, `Courier`, is used for characters that you type, or for lines of program code. `It looks like this`.

List of Abbreviations

The following abbreviations are used in this publication.

API	application programming interface
AV	audio/video
CD	compact disk
DLL	dynamic linked library
DITL	dialog item list
DM	Display Manager
SDK	Software Developer Kit
VCR	video cassette recorder

Other Reference Material

Related documentation includes the following books from the *Inside Macintosh* collection. *Inside Macintosh* is a collection of books, organized by topic, that describe the system software of Macintosh computers.The following publications can be found in the *Inside Macintosh* CD:

- *Inside Macintosh: QuickTime*
- *Inside Macintosh: More Macintosh Toolbox*
- *Inside Macintosh: Devices*

You should also refer to *Designing Cards and Drivers for the Macintosh Family,* third edition, published by the Addison-Wesley Publishing Company, Inc.

For More Information

APDA offers convenient worldwide access to hundreds of Apple and third-party development tools, resources, and information for anyone interested in developing applications on Apple platforms.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDA order
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

CHAPTER 1

Overview of the AV Architecture and Application Software

This chapter provides an overview of the AV Architecture: its capabilities, areas of application, and the technology that implements it. It also gives an overview of the windows and panels designed for applications based on the AV Architecture. These windows and panels replace the Macintosh standard Sound and Monitors control panels. They are used to configure all the audio and video (AV) devices connected to your Macintosh computer, or the AV features of the computer itself.

AV Devices

The audio devices referenced in this developer note include

- any device that provides sound input, such as microphones, CD players, videocassette players, audiocassette players, television tuners, telephone connections
- any device that receives sound and plays it back, such as speakers, headphones, videocassette recorders, audiocassette recorders

Video devices referenced include

- video input devices such as video cameras, videocassette recorders, television tuners
- video output devices or displays such as computer monitors, television sets, videocassette recorders.

Note
The devices listed are samples of the devices a typical AV application can control. The ones you can actually use depend on the computer or other AV product you are using. ◆

Why the AV Architecture Was Developed

This section provides background information about existing AV control panels, describes the problems that arise when these panels are used with new AV products, outlines the solution developed by Apple Computer, Inc. in the form of the AV Architecture, and finally touches on the ways developers can use this architecture.

Status Quo — Control Panels

Before the development of the AV Architecture, Macintosh computers used control panels, such as Monitors, Sound, and Video, to access and configure a variety of system devices, including audio and video (AV) devices. At the simplest level, you could access the Sound and Monitors panels from the Control Panels folder in the System Folder. Developers also created a variety of new and unique control panels to control their audio and video devices.

These panels allowed you to select and change audio and video features on your computer. For example, the Monitors control panel allowed you to select color or monochrome, identify the monitor, and set bit depth. The Sound control panel allowed you to select the source for sound input, change or mute the speaker volume, and select an alert sound. The options available within these control panels depended on the equipment connected to your computer.

Problem Areas

In a multimedia context, audio and video control panels proliferated as new AV technologies evolved. Control panels were developed to satisfy the needs of individual devices, such as audio/video displays with their various I/O (input/output) features. These control panels provided little centralization when a system was configured with multiple devices, they frequently duplicated functions, and it was sometimes necessary to go to two or more control panels to accomplish what seemed to the user to be one task. In addition, each panel had its own personality and was not aware of the inter-dependency between the technology it controlled and other technologies coexisting in the system. Users became confused by the number and variety of control panels.

Apple's Solution

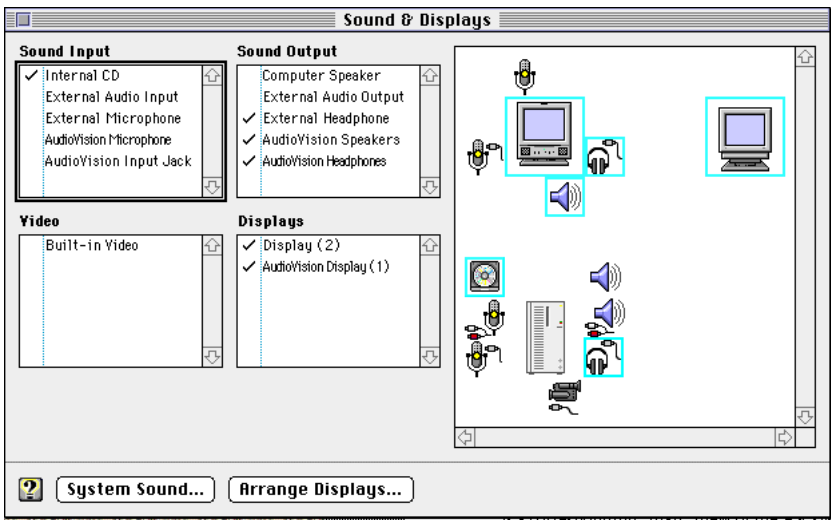
To deal with this proliferation, Apple Computer Inc. developed the AV Architecture. It provides a Macintosh framework that integrates all configuration features and allows you to develop AV applications with consistent user interfaces. Such applications enable you to access and configure the AV features of Macintosh computers and multimedia displays. They allow you to:

- select and modify the characteristics of AV input/output devices, such as speakers and CD players
- change display characteristics such as brightness and contrast, or other more advanced features

The AV Architecture's panels replace the Monitors, Sound, and Video control panels. The basic AV application currently provided by Apple is the Sound & Displays application. Sound & Displays provides a categorized front-end, for which users can access the individual panels that control AV devices. Figure 1-1 shows the Sound & Displays setup window.

If you are working with AV devices, you should develop AV components rather than control panels. Following the guidelines in this developer note, you should be able fit your product's panel seamlessly into the Sound & Displays application and into other future AV applications.

Figure 1-1 Sound & Displays setup window



The AV Architecture is easy to use, powerful, flexible, and extendable. It is implemented by the Component Manager, a dynamic linked library (DLL) technology that is now part of the Macintosh Toolbox. The Component Manager is described in detail in *Inside Macintosh: More Macintosh Toolbox*. The AV Architecture specifies a framework for separating the following elements into standard, reusable Component Manager components, such as:

- the driver-like software elements, or engines, that interface with audio and video hardware devices
- the user interface elements, or panels, that use the engines to control the audio and video devices

By separating elements of the AV software in this way, you can easily create new panels, update panels for existing engines, and reuse existing panels for new engines.

Architectural Features

The AV Architecture consists of the following components:

- Panel Components
- Engine Components
- Port Components
- Device Components
- the optional Manager Component

These components enable you to develop the user interface panels and functionality needed to communicate with all the AV devices connected to your Macintosh computer. As a developer you may design your own components, however, you must follow the baseline architecture as described in this developer note. You can also customize or override existing components. You may need to create only Panel Components, or you may need to create all five types of components. The component types are described in this document in the order you are most likely to need to create them.

The architecture incorporates the functionality essential to current and future projects and performs the following functions:

- Provides a foundation for a new AV user interface and allows users to access technological capabilities, such as video mirroring and double buffering, that will be available in upcoming products.
- Supports localization, which is the process of adapting software to a particular region, language, and culture. Elements involved include date and time format, keyboard resources, and fonts.
- Supports systems that are not audio/video systems.

Using the AV Architecture you can develop applications that

- support scripting and Apple events
- are dynamic and configure themselves based on the capabilities of the system on which the application is running
- provide a clear conceptual model for configuring AV devices
- create as few layers of interaction as possible and avoid modal dialog boxes where possible
- support task-oriented configurations, such as “play audio CD,” “enable microphone,” and so forth
- access frequently used settings easily and quickly
- implement configuration changes immediately, so that if you change brightness or contrast using a Video panel, you will see the changes reflected immediately on the screen

- provide supplementary visuals when needed
- have comprehensive on-line assistance through the Apple Guide on-line help system
- allow system settings to be changed automatically through **Apple events** and through scripting and recording.

Applications normally interact with Panel Components, Port Components, and the Macintosh Toolbox.

When you configure AV features, you activate and deactivate the I/O devices, as well as adjusting the various audio and video settings. You do this by means of an AV setup panel. Figure 1-1 on page 4 shows the setup window used in the Sound & Displays application.

Developer's Role

As a developer, you can use the AV Architecture in a number of ways. For example you can:

- Develop new Panel Components and accompanying Engine Components that will allow users to control your AV devices using Sound & Displays. Apple supplies this application routinely with products that use the AV Architecture.
- Develop new Engine Components that can be used by standard existing panels to control your AV devices.

In the future, you will be able to:

- Incorporate new or existing AV components into your own application. For instance, you might invoke a standard or new panel from within an audio application to control the volume of the CPU's audio output.

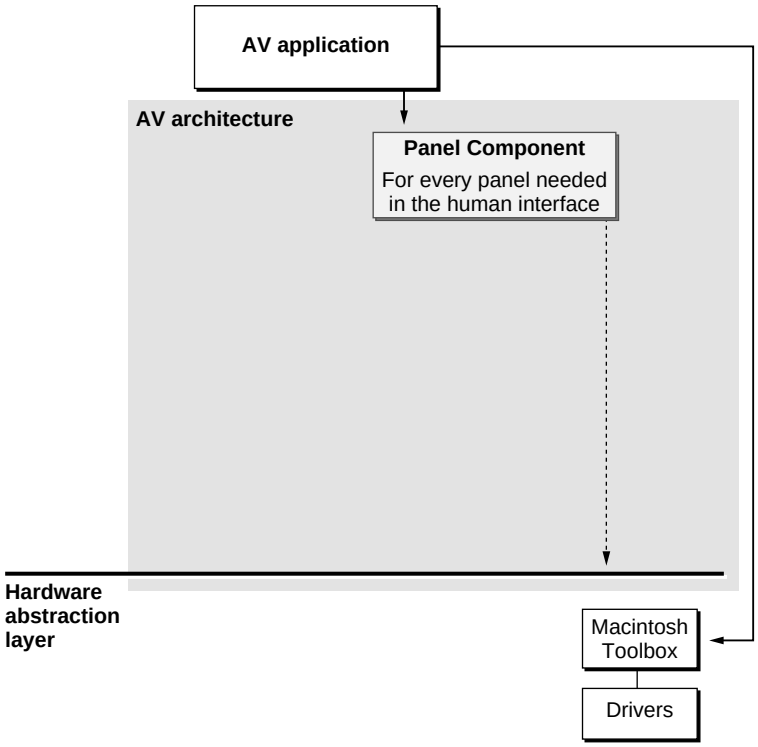
Architectural Components

This section gives an overview of the architectural components. It explains the main responsibilities of each component, what it does, and why it exists; it describes how to deliver components; and finally it looks at the philosophy behind a component-based architecture. Subsequent chapters of the developer note provide detailed information about each component.

Panel Components

As shown in Figure 1-2, Panel Components are at the highest level in the AV Architecture. You require a Panel Component for every panel needed in the human interface. The display shown in Figure 1-1 on page 4 has four panels: Sound Input, Sound Output, Video, and Displays.

Figure 1-2 Panel Components



IMPORTANT

The panels shown in this section are examples of typical panels from several AV applications including Sound & Displays. The panels you see in your own system environment will be different and will reflect the AV devices and the AV application you are using. ▲

Panel Components have the following general characteristics:

- allow you to provide a user interface
- contain no hardware-specific code
- do not deal with bits and bytes or register-level transactions
- have no knowledge of the hardware implementation
- generally focus on a single task or set of controls, such as bit depth or resolution

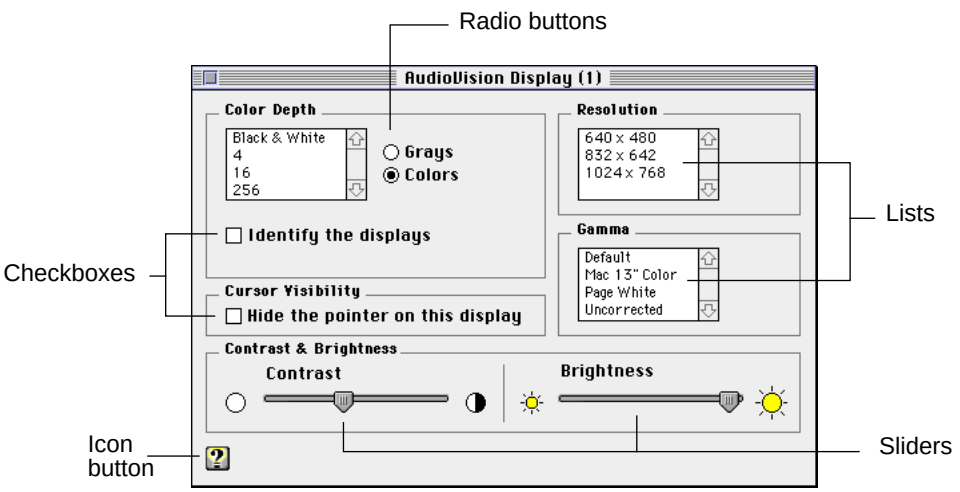
Panel Components perform the following specific functions:

- get the items in the **DITL** (dialog item list)
- get the title of the panel
- handle user interaction events

Panel Components generally access the hardware by means of the Engine Component, as described in the following section. A given panel can access more than one engine, and more than one panel can access the same engine. Panel Components may also use the services of the Macintosh Toolbox, as shown in Figure 1-2. They may even use drivers if the devices are available. They need know nothing about Port Components or Device Components.

Figure 1-3, a window containing five panels, shows some of the elements that commonly make up a panel.

Figure 1-3 Typical control panel elements



Panel elements include

- sliders that allow you to adjust things like contrast and brightness
- checkboxes that allow you to turn features on and off, for example displaying or hiding the pointer, and identifying the display
- radio buttons that allow you to select among features, for example, geometry features such as height/width, position, or, as shown in Figure 1-3, to select grayscale or color.
- lists that allow you to display and select among features such as the gamma features Default, Mac 13” Color, Page White, and Uncorrected

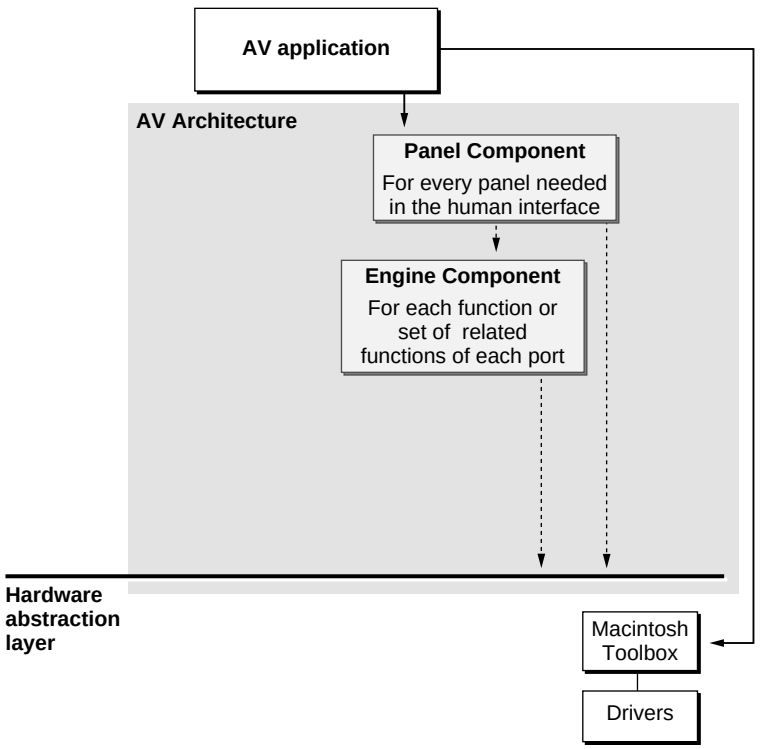
Panel elements not shown in Figure 1-3 include pop-up menus that allow you to select from a group of items or features, and static text such as an unchangeable text string that contains the name of the display. Chapter 2 provides detailed information about Panel Components.

The human interface guidelines applied to AV applications call for panel designs that exclude Cancel or OK buttons. These buttons are associated with dialog boxes that you dismiss when you have finished using them. On the other hand, AV panels are not dismissed, but remain on the screen until you close the application. Since configuration changes initiated through AV panels take place instantly, if you make a change and then decide not to accept it, you can immediately go back and choose another setting.

Engine Components

Engine Components provide the Panel Component with access to the hardware, as shown in Figure 1-4. Note that for each function or set of related functions associated with a given port, you must supply individual Engine Components. So, for example, if you have a display port, you will need separate Engine Components for Contrast, Geometry, and VPT (**Virtual Photometry Technology**).

Figure 1-4 Engine Components



Using Engine Components to provide this kind of access has the following advantages:

- Specific hardware elements are separated from the user interface.
- Engine Components act as a sort of **hardware abstraction** layer, and provide common APIs (application programming interfaces) to hardware functionality. These are high-level APIs that are easy to use and contain calls such as `GetBrightness()` and `SetContrast()`, rather than `SetDeviceRegister`.
- Engine Components contain the mechanism for controlling specific attributes, such as the Geometry functionality of a display port.
- Several development teams can work independently on different elements of the software at the same time. For example, some developers may be working on hardware drivers while others are developing human interface panels for the user interface.

- Elements such as Contrast or Brightness are quite different from each other and share only the mechanism for communicating with the hardware. Because the mechanism for controlling specific element functionality is encapsulated in an Engine Component, the architecture is more modular, and specific engines can be upgraded without disturbing other engines.
- If you are viewing only the human interface of a specific element, such as Brightness, the code for the other elements, such as Contrast, need not be in memory at the same time. This type of code may be voluminous and viewing elements in this way saves memory space.

Refer to Chapter 3 for further information about Engine Components.

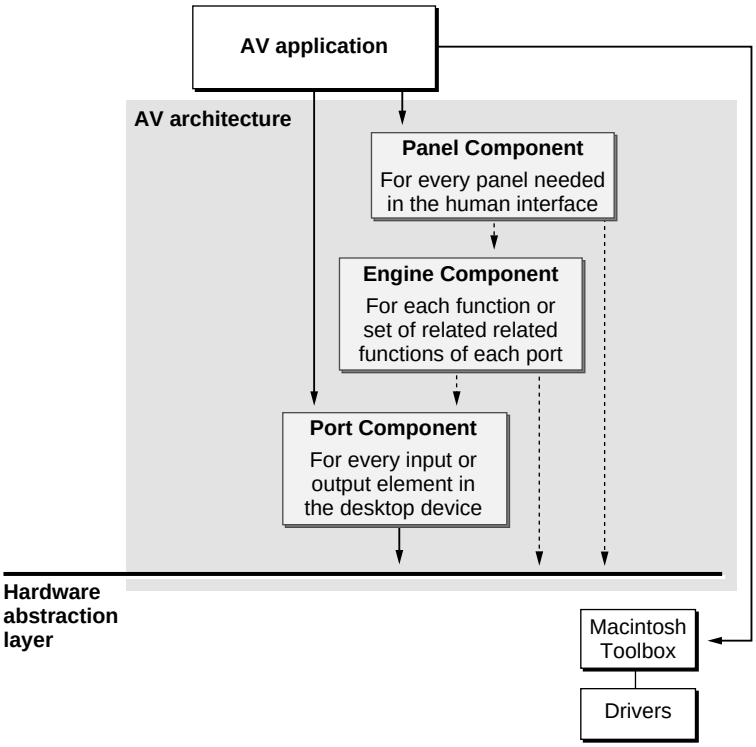
Port Components

Each AV product has a port for each audio or video I/O element. Port Components group control panels by their functionality and are defined in terms of type (category), port, physical element, and attributes, as shown in Table 1-1. Figure 1-5 shows how Port Components relate to other components in the AV Architecture.

Table 1-1 Port Component definition

Type or category	Port	Physical elements	Attributes
Audio input	Microphone port	Microphone CD player	Gain Playthrough Balance
Audio output	Speaker port Headphone port	Speakers Headphones	Volume Balance
Video input	Video port	Camera VCR	Inputs Format Filter
Video output	Display port	Monitor Display Television screen VCR Camcorder	Contrast Brightness

Figure 1-5 Port Components



Port Components contain calls that return the name and image of the port, and they provide access to the hardware. In addition, they can turn an individual piece of hardware, such as the headphone port, on and off. Access may be directly to the hardware, using calls such as `SetDeviceRegister()` and `GetDeviceRegister()`.

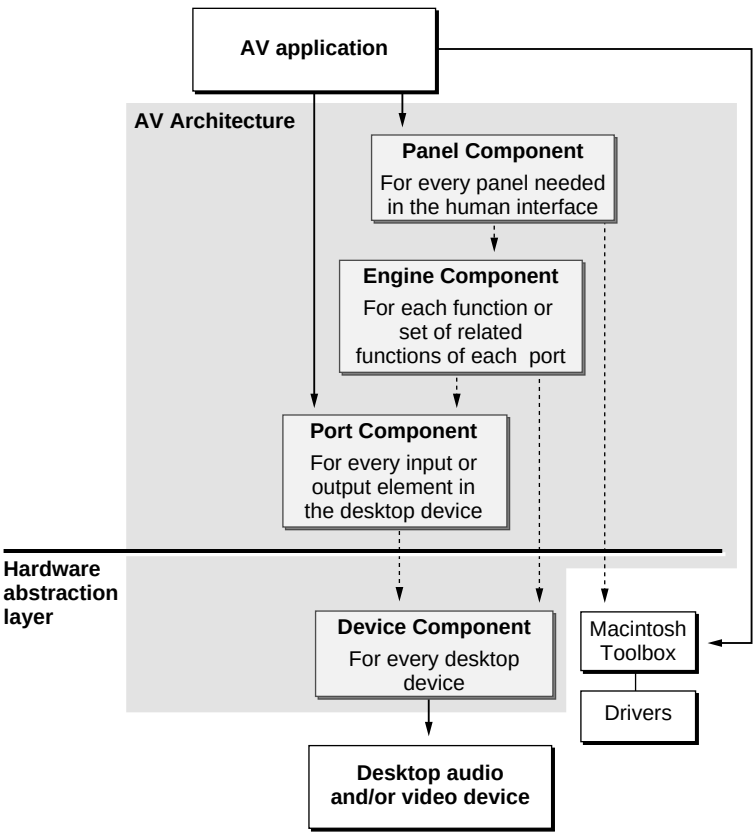
Port Components also help the Display Manager to detect and identify displays connected to the system. “Implementing Wiggling” on page 42 provides more information on detecting and identifying hardware elements on ports that have more than one communication/control path.

Refer to Chapter 4 for further information about Port Components.

Device Components

Device Components represent the collection of features or ports associated with a particular physical device. A device can be a single display with one port; or a display, such as the AppleVision 1710AV, with multiple ports and complex audio/video features; or a CPU with audio capabilities. Figure 1-6 shows the position of Device Components in the architectural hierarchy.

Figure 1-6 Device Components



All Device Components perform the following functions:

- They contain a call that returns the name of the device, such as AudioVision, or AppleVision.
- They contain a picture (icon) of the device that can be displayed in a graphical representation of the control panel or application.
- They can act as drivers, with an expanded API that has additional calls such as `SetDeviceRegister()` and `GetDeviceRegister()` for communicating directly with the device's hardware.

Note
Port Components can also act as drivers. However, if more than one port shares a common communication mechanism, it makes sense to use the Device Component to implement this functionality. ◆

Refer to Chapter 5 for further information about Device Components.

Delivering Your Components

The components that you provide should be built into one or more system extensions. The components can be encapsulated in the extension(s) in one of several ways:

- as separate 'thng' files, each containing a separate AV component
- as one 'thng' file containing all your AV components
- as an INIT, which registers one Manager Component, which in turn registers all your other AV components

You must use option two or option three if your components need to load in a predetermined order. Appendix A provides more information on the Manager Component. The "Component Manager" chapter of *Inside Macintosh: More Macintosh Toolbox* provides information about 'thng' files. You should also refer to *develop* Issue 15, and the article "Managing Component Registration" for more information about various methods of registering Component Manager components. You will find *develop* in the Periodicals folder on the CD ROM, *Developer CD Series Reference Library*, December 1994.

Advantages of a Component-Based Architecture

After looking at the different components that are part of the AV Architecture, it becomes apparent why the architecture is component based. Like object-oriented architectures, the AV Architecture is dynamic and modular. It allows you to replace or redesign individual components without replacing the entire structure. This means that individual functionality can be customized; new features, such as Contrast and Brightness, can be changed and added; troubleshooting can be done at the component level, making it easier to identify and fix problems.

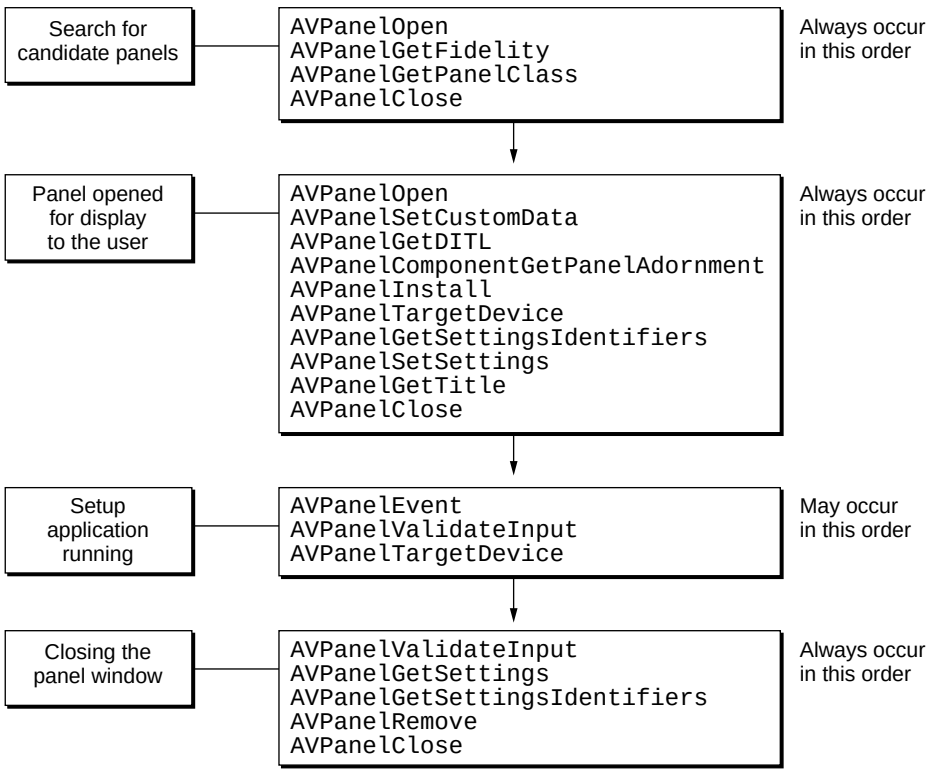
In addition, separating Engine Components from Port Components allows each port to be upgraded independently. It also allows the Engine Components to be used on newer devices and even on different hardware with different Port Components, provided that the API to the port is kept constant.

CHAPTER 2

Panel Components

This chapter describes the sequence of calls to each of the Panel Component functions. When a panel window is being created, the chain of calls to AV Panel Components is similar to the call sequence shown in Figure 2-1. This chapter describes how the panel owner searches for candidate panels, opens candidates for display to the user, runs the setup application, and then closes the panel window.

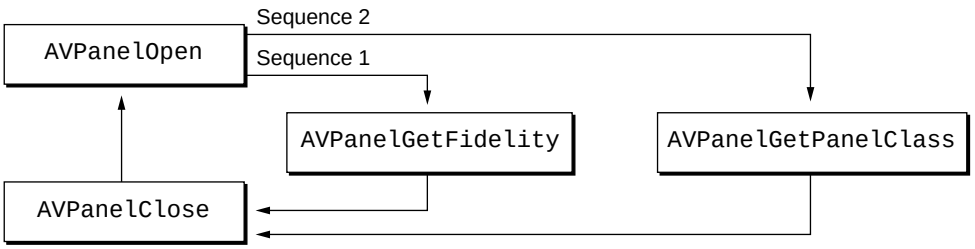
Figure 2-1 Overview of call sequence



Panel Owner Search for Candidate Panels

When the panel owner, which may be the AV application, searches for candidate panels to display to the user, it usually asks the Display Manager to provide a list of candidates. During this search, the calls shown in Figure 2-2 are made. During the first part of the sequence the panel is opened, and a fidelity check is made to see how well the panel functions on the specified port. If you get an acceptable fidelity measurement, the panel closes and then reopens, and the call is made to get the specific class of the AV functionality. The class may be sound, geometry control, color correction, and so forth. If the fidelity measurement is not acceptable, indicating that the panel generally does not function with the given port, the panel closes and is not reopened.

Figure 2-2 Call sequence—panel owner search for candidate panels



IMPORTANT

The functions are described from the panel viewpoint. The parameters to the functions are not the same as the parameters to the functions called by the controlling application. This is because the Component Manager translates a component selector call from the format of the calling function, to the format of the function being called. ▲

AVPanelOpen

`AVPanelOpen` is a standard component function, as described in standard Component Manager documentation.

```
pascal ComponentResult AVPanelOpen (Handle storage,
                                    ComponentInstance self)
```

`storage`	The handle that was associated with the panel by the `SetComponentInstanceStorage()` call.
`self`	An instance of your panel component.

Supplementary Information

This panel should perform the following functions as a minimum:

- It should decide whether it is safe to open another instance of the component. Some AV panels restrict themselves to having only one instance open at a time, while others allow an unlimited number of instances to be open.
- `AVPanelOpen` must also allocate a handle to hold the instance globals. Refer to "Creating and Destroying Common Globals" on page 57 for information on this subject. There is a single data structure that a panel defines for itself. This structure holds all the information needed for the panel to manage itself. The handle allocated by the panel is equivalent to the size of the data structure.
- The panel initializes the `self` field of the instance globals. Panels are free to store whatever information they need. However, one of the fields of the instance globals data structure should be a `Component` type. This field should be initialized to hold the `self` parameter that was passed into the `AVPanelOpen()` routine. It is useful to have this information available at certain times.

- The panel sets the component instance storage. In order to have the instance globals handle passed back each time one of the component selectors is called, the panel must call `SetComponentInstanceStorage (self, (Handle) globals)`. Doing this allows the panel to associate the newly allocated handle with the panel instance.

Sample Code: AV Architecture

In the following example, the variable `globals` was the local parameter used to allocate the handle.

```
pascal ComponentResult
AVPanelOpen (Handle storage, ComponentInstance self)
{
#pragma unused (storage)

// NOTE: This code assumes the definition of a data structure
//   called MyPanelGlobals, a data type called
//   MyPanelGlobalsHdl, and a constant called
//   kMaxNumberofMyPanelInstances

ComponentResult   result = noErr;
MyPanelGlobalsHdl  globals;

SetComponentInstanceA5 (self, (long ) LMGetCurrentA5());

// Can we open another instance?

if (CountComponentInstances ((Component) self) <=
                    kMaxNumberOfMyPanelInstances)
{
         // Allocate our storage:

      globals = (MyPanelGlobalsHdl) NewHandleClear (sizeof
                                        (MyPanelGlobals));

      if (globals != nil)
      {
            // Keep a reference to self:

         (*globals)->self = (Component) self;

            // Set storage ref:

         SetComponentInstanceStorage (self, (Handle) globals);
      }
      else// NewHandleClear failed
```

```
      {
         result = MemError();
      }
   }

   else// No more instances can be opened
   {
      result = kAVOpenComponentInstanceError;
   }
   return (result);
}
```

AVPanelGetFidelity

This function allows the panel to tell the caller how well the panel functions on a particular port.

```
pascal ComponentResult AVPanelGetFidelity
                              (ComponentInstance panelComponent,
                               AVIDType portID,
                               DMFidelityType* panelFidelity);
```

`panelComponent`	An instance of the panel component.
`portID`	The ID of the port for which fidelity is being returned.
`panelFidelity`	A number passed back to the caller that measures how well the panel supports the port in question.

Supplementary Information

During the process of searching for a candidate, the Display Manager opens the Panel Component and call its `AVPanelGetFidelity` function. At this point, the panel should examine the port ID that is passed in and determine how well it supports that port. Typically, if a good fidelity number is returned, the parent application will probably call `AVPanelTargetDevice` later and then open the panel for that port.

If a panel, such as a bit depth panel, needs no engines but instead needs a gDevice, it can just call:

```
DMGetGDeviceByDisplayID(portID);
```

If no gDevices come back, it will return `kNoFidelity`.

If the panel needs one or more engines, it will search for the available engine type(s) it needs by making the following call for each engine type:

```
DMNewAVEngineList (portID, engineType, minimumFidelity,
                   engineListFlags,reserved, engineCount,
                   engineList);
```

The parameters to this call are as follows:

`portID`	The ID of the port for which fidelity is being returned.
`engineType`	This is the component subtype of the particular kind of engine for which you are searching. For example, if the Brightness panel needs to find its engine, you would provide the component subtype of your brightness engine here. Note that there may be more than one engine registered with this subtype. Some of these engines may not belong to you.
`minimumFidelity`	This parameter sets the minimum level of fidelity you will accept. The interfaces shipped with the SDK (Software Developer Kit) enumerate the constants used to define the levels of fidelity.
`engineListFlags`	Currently you should set this parameter to `0`.
`reserved`	Set this parameter to `0`.
`engineCount`	When this parameter is returned, it contains the number of engine components that responded to this call.
`engineList`	This is the list of engine components. To get the relevant information about each engine, you can use an engine iterator function to cycle through the list.

There may be several engines of `theType`. The Display Manager will find all registered engine components that match the criteria to the function, and it will call their `AVGetFidelity` functions. The returned engine list contains a reference to each of the engines that succeeded in responding to this query. If there is more than one engine in the returned list, your panel will need to query each one to find the one that the panel expects. One way to do this is have one or more selector functions that your engine and panel know about. Using this communication mechanism, your panel can determine whether it has found the engine it needs.

The `AVPanelFidelity` value that your component passes back from this function should be set based on the results of the steps taken:

- If the panel determines that it has the ability to control the specified port completely by means of the port ID (`portID`), it can return `kManufacturerFidelity`.
- If the panel determines that it is not the owner of the port, but it knows how to control the functions of the port, it can return `kDefaultFidelity`.
- If the panel determines that it knows nothing about how to control the port, it can return `kNoFidelity`.

AVPanelClose

This function is a standard component function, as described in the Component Manager documentation.

```
pascal ComponentResult AVPanelClose
                        (Handle storage, ComponentInstance self)
```

`storage`	The handle that was allocated in the `AVPanelOpen()` routine and that was associated with the panel by the `SetComponentInstanceStorage()` call.
`self`	An instance of your panel component.

Supplementary Information

The panel should be able to do the following:

- Dispose of any data handles or component references that are referenced from this panel's instance globals.
- Dispose of the instance globals handle.

AVPanelGetPanelClass

Apple Computer defines standard classes of AV functionality, such as sound, geometry control, color correction control, standard video/graphics control, and so forth. Classes allow you to use one method of grouping and displaying panels to the user.

```
pascal ComponentResult AVPanelGetPanelClass
                    (Handle storage, resType* panelClass,
                     resType* panelSubClass, Ptr reserved1,
                     Ptr reserved2)
```

`storage`	Your panel's storage data, created in the `AVPanelOpen` routine.
`panelClass`	A pointer to the `resType` which receives the class of your panel. This is either one of the pre-defined constants (see the sample code, page 18 for the definitions), or a class of your own definition.
`panelSubClass`	This is also a pointer to a `resType`. It is currently not used, so you should set it to `nil`.
`reserved 1` and `2`	Set these parameters to `nil`.

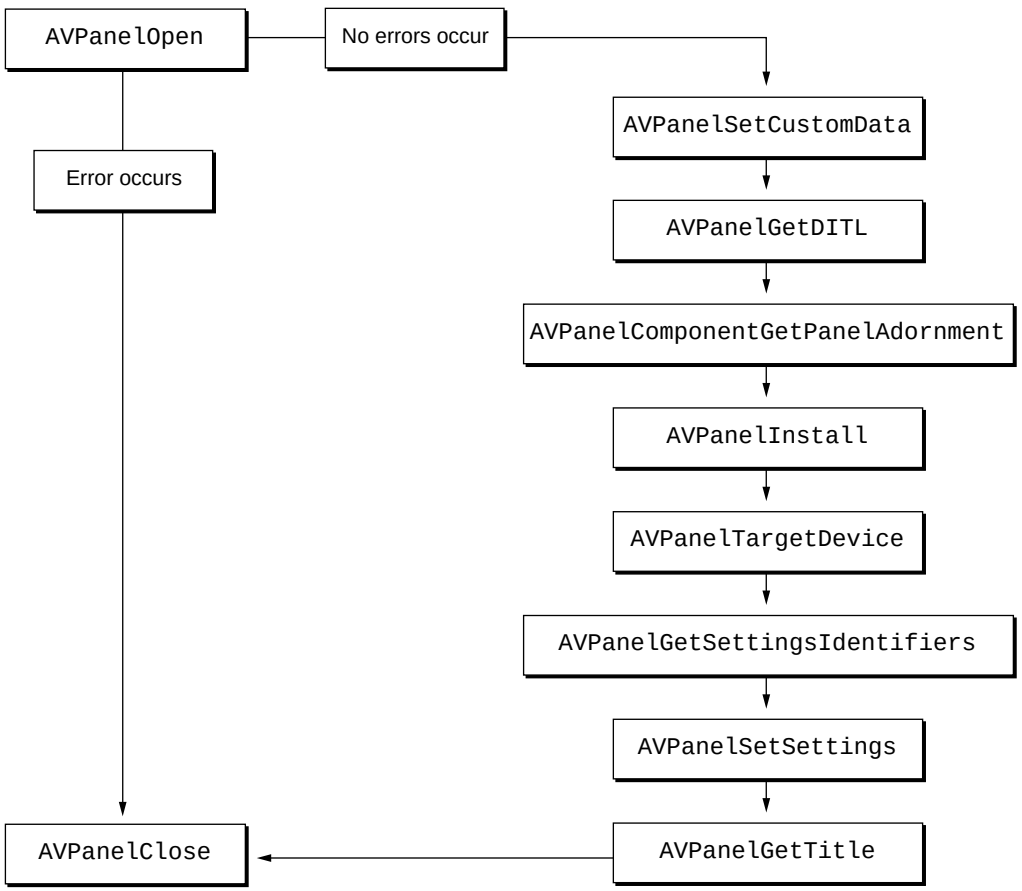
Supplementary Information

There is no supplementary information about this function.

Opening Panels for Display to Users

Selected panels are opened for users in the sequence shown in Figure 2-3. If an error occurs while the panels are being opened, a call is made immediately to `AVPanelClose`.

Figure 2-3 Opening panels for display to users



AVPanelOpen

The `AVPanelOpen` function is described on page 17, and it works in exactly the same way for this sequence.

AVPanelSetCustomData

This function allows two components to share specific data. It enables one component to share information with another component that is being launched. The entity that creates the parameters to launch a given panel has the option of specifying the custom data, usually as a handle to a later structure.

```
pascal ComponentResult AVPanelSetCustomData (Handle storage,
                                             long theCustomData)
```

`storage`	The handle that was allocated in the `AVPanelOpen()` routine and that was associated with the panel by the `SetComponentInstanceStorage()` call.
`long`	A long integer.
`theCustomData`	Field of data structure.

Supplementary Information

There are occasions when one panel may need to create and launch a separate panel window. The process of doing this involves creating a variety of data structures that tell the panel window what panels should be in the window. One of the fields of one of the data structures is the custom data field. This is a long integer that can be treated as a standard refCon-type field. The function is a mechanism that allows a panel to pass specific data to the panel that is being launched. The data may be

- a four-byte value
- a handle to a data structure that both of the panels can manipulate
- the component reference of the panel that is causing the new panel window to be launched

AVPanelGetDITL

This function allows the parent application to determine which dialog box items are managed by your panel. The parent application uses this information to build the configuration window for users.

```
pascal ComponentResult AVPanelGetDITL
               (ComponentInstance panelComponent, Handle *ditl);
```

`panelComponent`	An instance of the panel component.
`ditl`	Refers to the `Handle` that is to receive the component's dialog item list. The component should allocate a handle to the DITL, either through a `GetResource()` call, or by manually creating a DITL and returning this handle.

Supplementary Information

The parent application calls your `AVPanelGetDITL` function to obtain the list of dialog box items supported by your panel. The panel then puts these items into a window and presents the window to users.

The component returns the item list in a `Handle` provided by the panel.

Note
The parent application disposes of this `Handle` after retrieving the item, so you should make sure the item list is not stored in a resource. If your item list is already in a resource handle, you can use the Resource Manager's `DetachResource` function to convert that resource `Handle` into one that is suitable for use with the `AVPanelGetDITL` function. ◆

The parent application will open your resource file before calling the function, unless you instructed the parent application not to open your resource file by setting the `channelFlagDontOpenResFile` component flag to `1`.

Note
If the panel needs to add any items to the DITL dynamically, this is the time to do it. The DITL handle should be resized appropriately, and the necessary items inserted into the structure. ◆

Note
You will have the opportunity to adjust the DITL items in the `AVPanelTargetDevice` function. ◆

AVPanelComponentGetPanelAdornment

This function allows the panel to request how it will be adorned when it is displayed in the panel window. A panel's adornment consists of the border and the name.

```
pascal ComponentResult AVPanelComponentGetPanelAdornment
      (Handle storage, long *panelBorderType, long *panelNameType)
```

`storage`	The handle that was allocated in the `AVPanelOpen()` routine and that was associated with the panel by the `SetComponentInstanceStorage()` call.
`long`	A long integer

Supplementary Information

There is no supplementary information about this function.

Sample Code

This section shows how a panel should request to be adorned. The options currently available for adornment are as follows:

```
enum {
      kAVPanelAdornmentNoBorder = 0,
      kAVPanelAdornmentStandardBorder
};

enum {
      kAVPanelAdornmentNoName = 0,
      kAVPanelAdornmentStandardName
};
```

Figure 2-4 shows an example of a panel that has both name and border displayed. Figure 2-5 shows a panel with a name but no border. Figure 2-6 shows a panel with no name and no border. The text "AudioVision 14 Display" shown in Figure 2-6 is just a text string that is part of the panel. It is not the name of the panel.

Figure 2-4 Panel with name and border

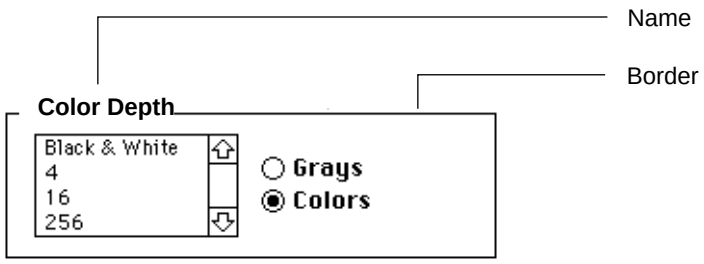


Figure 2-5 Panel with name and no border

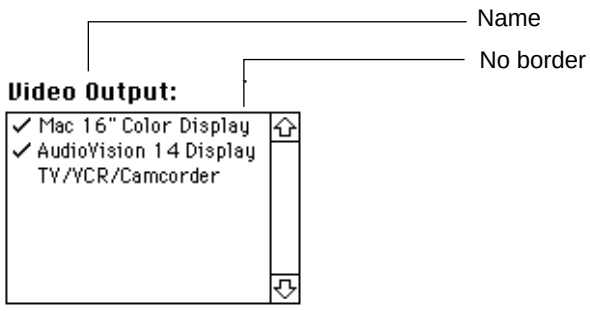
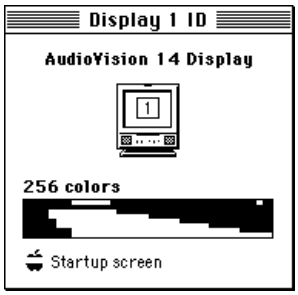


Figure 2-6 Panel with no name and no border



AVPanelInstall

The parent application calls the `AVPanelInstall` function after adding your items to the window. This routine is called only once throughout the life of the panel, so the operations performed should be actions that need to be taken only once.

```
pascal ComponentResult AVPanelInstall
                            (ComponentInstance panelComponent,
                             DialogPtr theDialog,
                             short itemOffset);
```

`panelComponent`	An instance of the panel component.
`theDialog`	A pointer identifying the panel's dialog box.
`itemOffset`	Specifies the offset to the panel's first item in the window. Because the parent application builds your dialog box items into a larger window containing other items, this value may be different each time your panel is installed. Do not depend on it being the same.

Supplementary Information

The parent application provides the panel with the information that identifies the window and the offset of the panel's items in the DITL. Since this function is called only once, you may use this opportunity to do the things that need to be done only once. This might include creating a new empty list handle if any of your DITL items are lists; positioning items within your panel's boundaries if they need to be adjusted; loading resources that are not specific to any particular port or device, and saving them in the instance globals.

IMPORTANT

At this time, the panel does not have any knowledge of the specific port it will be asked to control, so there are limitations to the kinds of things that can be accomplished. For example, the Bit Depth panel can allocate an empty list, but it cannot fill in the list entries with the bits depths supported for a given port until it knows which port it will be controlling. This information becomes available through the `AVPanelTargetDevice()` function. ▲

AVPanelTargetDevice

This function controls specific ports and the device referred to is a port. The function may be thought of as the continuation of `AVPanelInstall()`. However, unlike `AVPanelInstall()`, it may be called more than once during the life of the panel.

```
pascal ComponentResult AVPanelTargetDevice
                    (Handle storage, AVIDType displayID,
                     DialogPtr theDialog, long itemsOffset)
```

`storage`	The handle that was allocated in the `AVPanelOpen()` routine and that was associated with the panel by the `SetComponentInstanceStorage()` call.
`displayID`	The AV ID (audiovideo ID) of the port that is being targeted.
`theDialog`	The dialog box window that owns this panel.
`itemsoffset`	The offset into the DITL of the window for this panel's items.

Supplementary Information

The targeting action of this function gives the panel an opportunity to configure itself based on the specific capabilities of the port it will be controlling. If the panel requires an engine to manipulate the hardware, it can find and open an instance of the appropriate engine, as it did during the `AVPanelGetFidelity()` function. However, this time it should save a reference to the Engine Component in its globals, for use as the panel is receiving events from the user.

The panel can get the current state of the hardware by making engine (or Macintosh Toolbox/driver) calls. It can then set its controls and globals fields appropriately.

The important thing to remember about this function is that it can be called multiple times while the panel is open. The AV Architecture is designed this way to allow flexibility in the human interface design of AV applications. For example, an application may wish to keep a pop-up menu of the monitors connected to a Macintosh computer. When the user chooses a new item from the pop-up menu, the application simply asks the existing panels to target themselves to this new port.

Because targeting can happen multiple times, it is important that you do not rely on the state of the hardware-specific globals fields that were not initialized in the `AVPanelInstall()` routine. For example, if some of the fields have significance when their value is `0`, and the globals handle was originally allocated with a `NewHandleClear()` toolbox call, it may seem safe to assume that you can rely on them being clear after the panel leaves the targeting routine. However, this is not true, since the panel may have been running and modifying the fields of the globals and was subsequently asked to target again. It is important to set any hardware-specific fields in the globals that need to be initialized when the targeting routine is complete, in case there are other functions in the panel component that rely on this information.

AVPanelGetSettingsIdentifiers

The parent application calls the component's `AVPanelGetSettingsIdentifiers` function to get the identifier (number) and the type under which your settings are stored. The identifier is an integer. The type is a four-character resource ID that fits into the long word format. The identifier and type uniquely identify the information that will be stored to disk.

```
pascal ComponentResult AVPanelGetSettingsIdentifiers
                        (ComponentInstance panelComponent,
                         short *theID, OSType *theType);
```

`panelComponent`	An instance of your panel component.
`theID`	The identifier under which your settings should be stored.
`theType`	The type under which your settings should be stored.

Supplementary Information

There is no supplementary information about this function.

AVPanelSetSettings

The parent application calls the component's `AVPanelSetSettings` function to restore the panel's settings using previously saved values.

```
pascal ComponentResult AVPanelSetSettings (Handle globals,
                                           Handle ud, long flags);
```

`globals`	The panel's `Handle` for global data.
`ud`	Identifies a `Handle` that contains new settings information for your panel. Your component must not dispose of this `Handle`.
`flags`	Reserved for future use.

Supplementary Information

The parent application calls the `AVPanelSetSettings` function to restore your panel's settings. It may use this call to set default values before displaying the panel to the user.

Your component originally creates the settings information when the parent application calls the `AVPanelGetSettings` function. The parent application passes this configuration information back to you in the `ud` parameter to the `AVPanelSetSettings` function. Your component should parse the configuration information and use it to establish your panel's current settings.

Note
Your component may not be able to accommodate the original settings. For example, settings may have been stored for some time, and the hardware environment may not be able to support the values in the settings. You should try to make the new settings match the original settings as closely as possible. If you cannot make a perfect match, return an appropriate result code. ◆

Note
Do not save settings that are managed by the system, such as bit depth, resolution, and so forth. Configure the panel to reflect these settings during the `AVPanelTargetDevice` routine. ◆

The parent application uses the component's `AVPanelGetSettings` function to retrieve this configuration information.

AVPanelGetTitle

The parent application calls the `AVPanelGetTitle` routine if it wants to obtain the title of the panel. The parent application uses this string whenever an identifier for your panel is required. The identifier is used, for example, in the border title of a panel in the panel window.

```
pascal ComponentResult AVPanelGetTitle (Handle globals,
                                        StringPtr title);
```

`globals`	The panel's `Handle` for global data.
`title`	Refers to the `StringPtr` to hold your title. Fill in this `StringPtr` with the string you want to represent the title of your AV panel.

Supplementary Information

This routine is used to get the title of a panel. The parent application then uses this title, which is a string, to identify the panel to the user. The title is generally displayed in the panel border. The component returns the title in the `StringPtr` that is provided by the parent application.

The parent application opens the resource file before calling the function unless you have instructed the parent application not to open the resource file. That is, you have set the `channelFlagDontOpenResFile` component flag to `1`. This string could be stored in the resource fork of your component.

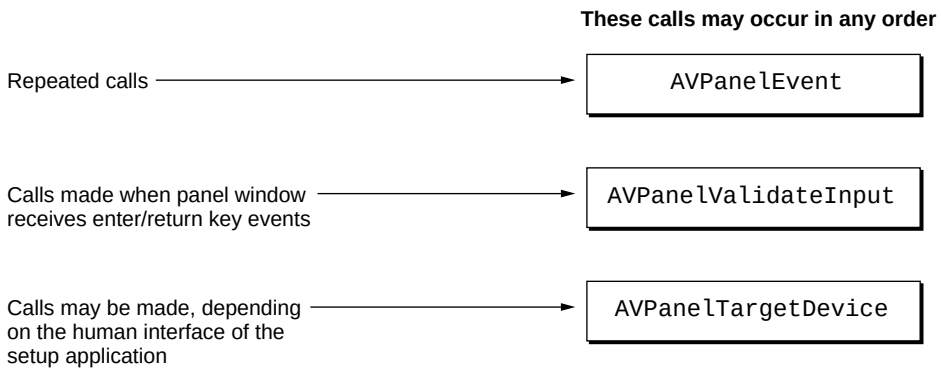
AVPanelClose

The `AVPanelClose` function is described on page 21, and it works in exactly the same way for this sequence.

Running the Setup Application

When the setup application is running and interacting with the user, calls are made to the functions shown in Figure 2-7 and described in this section.

Figure 2-7 Call sequence—setup application running



AVPanelEvent

The `AVPanelEvent` function allows your component to receive and process individual dialog box events. This function is similar to a modal dialog box filter function.

```
pascal ComponentResult AVPanelEvent(Handle globals, DialogPtr
                                    dialog, short itemOffset,
                                    Eventrecord *theEvent,
                                    short *itemHit,
                                    Boolean *handled);
```

`globals`	The panel's `Handle` for global data.
`dialog`	A dialog pointer identifying the settings dialog box.
`itemOffset`	Specifies the offset to the panel's first item in the dialog box.
`theEvent`	Contains an event record, which contains information identifying the nature of the event.
`itemHit`	Refers to a field that is to receive the item number in cases where the component handles the event.
`handled`	Refers to a Boolean value. Set this Boolean to indicate whether or not the component handles the event. Set it to `TRUE` if it handles the event, and to `FALSE` if it does not.

Supplementary Information

The parent application calls your `AVPanelEvent` function whenever an event occurs in the settings dialog box. The `AVPanelEvent` function is similar to a modal dialog box filter function. The main difference is that rather than returning a Boolean value to indicate whether or not the event was handled, the `AVPanelEvent` function sets a Boolean that is provided by the calling function. If you handle the event, be sure to update the field referred to by the `itemHit` parameter. By default, this value will be initialized to `true` by the parent application, so you need only be concerned with setting it to `false`. Some events, such as mouse down, focus on one panel. Others, such as update, activate, deactivate, suspend, and resume, go to all panels. When you handle a single-panel event, the value should be set to `true`. If you do not handle the single-panel event, the value should be set to `false`. The value is always set to `false` for events that go to all panels, regardless of whether or not they are handled.

AVPanelValidateInput

The parent application calls the `AVPanelValidateInput` function to allow you to validate the contents of the user window.

```
pascal ComponentResult AVPanelValidateInput (Handle globals
                                                Boolean *ok);
```

`globals`	The panel's `Handle` for global data.
`ok`	Contains a pointer to a Boolean value. You set this Boolean to indicate whether user settings are acceptable. Set it to `TRUE` if the settings are correct, otherwise, set it to `FALSE`.

Supplementary Information

The parent application calls the `AVPanelValidateInput` function to allow you to validate the settings chosen by the user. This is your opportunity to validate them in their entirety, including those for which you may not have received dialog box events or mouse clicks. For example, if your panel uses a `TextEdit` box, you should validate its contents at this time. Be sure to give the user some indication of how to fix the settings.

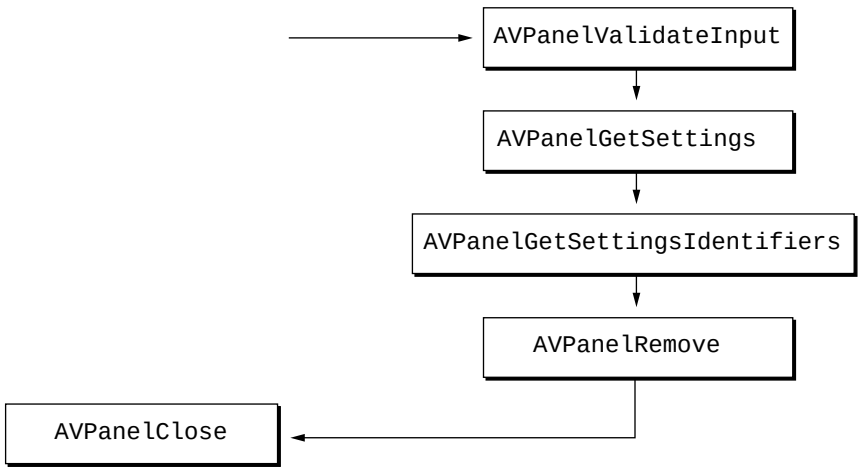
AVPanelTargetDevice

The AVPanelTargetDevice function is described page 27, and it works in exactly the same way for this sequence.

Closing the Panel Window

When you are ready to close the window that holds the panel, calls are made to the functions shown in Figure 2-8 and described in this section.

Figure 2-8 Call sequence—closing the panel window



AVPanelValidateInput

The AVPanelValidateInput function is described page 31, and it works in exactly the same way for this sequence.

AVPanelGetSettings

The parent application calls the component's `AVPanelGetSettings` function to retrieve the panel's current settings.

```
pascal ComponentResult AVPanelGetSettings (Handle globals,
                                           Handle ud, long flags);
```

`globals`	The panel's `Handle` for global data.
`ud`	Contains a `Handle` to your panel's configuration data. Your component is responsible for creating a new `Handle` and returning it as `ud`. Your component is not responsible for disposing of the `Handle`.
`flags`	Reserved for future use.

Supplementary Information

The parent application calls your `AVPanelGetSettings` function to obtain a copy of your panel's current settings. The parent application stores these settings and may use them to restore the panel's settings by calling the `AVPanelSetSettings` function. Your component should store whatever values are necessary to configure the associated panel component properly.

These settings may be stored as part of a larger panel configuration, and they must remain valid across system restarts. Therefore, you should not store values that may change without your knowledge, such as a component ID. In addition, saving a file reference number or component instance is not valid.

You are free to format the data in the `Handle` in any way you wish. Make sure you can retrieve the settings information from the user data item when the `AVPanelSetSettings` function is called. You may choose to format the data in such a way that other components can parse it easily, allowing the component to operate with other panels.

The parent application uses the component's `AVPanelSetSettings` function to restore this configuration information.

Note
Do not save settings that are managed by the system, such as bit depth, resolution, and so forth. Configure the panel to reflect these settings during the `AVPanelTargetDevice` routine. ◆

AVPanelGetSettingsIdentifiers

The function is described on page 28 and it works in exactly the same way for this sequence.

AVPanelRemove

The parent application calls the component's `AVPanelRemove` function before it removes the panel from the settings dialog box.

```
pascal ComponentResult AVPanelRemove(Handle globals,
                                     DialogPtr dialog,
                                     short itemOffset);
```

`globals`	The panel's `Handle` for global data.
`dialog`	Contains a dialog pointer identifying the settings dialog box.
`itemOffset`	Specifies the offset to the panel's first item in the dialog box.

Supplementary Information

The parent application provides you with the information that identifies the dialog box, and the offset of the panel's items into the dialog box. You may use this opportunity to save any changes you have made to the dialog box or to retrieve the contents of `TextEdit` items. If the parent application opened your resource file, it will still be open when it calls this function.

The `AVPanelRemove()` routine is the place to get rid of any data that pertains to the panel's membership in the panel window from which it is being removed. This is different from getting rid of the `globals` handle in `AVPanelClose()`, because the panel still exists after the `AVPanelRemove()` function has been completed.

You can use this routine, for example, to

- remove the display event notification registration, if one was created, by calling `DMRemoveExtendedNotifyProc()`
- remove any Apple event handler routines by calling `AERemoveEventHandler()`

You use `AVPanelRemove` to perform these functions because once the panel does not belong to a panel window any longer, it has no need to receive notification or Apple events. Any other ties to the system that are relevant only when this panel is installed should be removed at this time.

AVPanelClose

The `AVPanelClose` function is described on page 21, and it works in exactly the same way for this sequence.

CHAPTER 3

Engine Components

When Engine Components are required to operate a Panel Component, they are opened and closed by the Panel Component, usually when the panel itself is opened and closed.This chapter describes the Engine Component functions that are part of the basic AV Architecture, it describes how to register Engine Components multiple times, and also explains how notification takes place. Refer to “Engine Components” on page 9 for an overview of Engine Component functions and the functional advantages associated with Engine Components.

Engine Component Description

Engine Components have the following component description:

`componentType = 'avec';`	Indicates whether the component is an audio or video panel.
`componentSubType = '????'`	Identifies the specific engine.
`componentManufacturer = '????';`	Identifies the manufacturer.
`componentFlags = cmpWantsRegisterMessage;`	Makes sure the component is suitable for the machine.

Engine Component Functions

This section describes the Engine Component functions used in the AV Architecture .

AVEngineComponentGetFidelity

This function allows your engine to tell the caller (typically a panel with its own `GetFidelity` function) how well your engine can perform on a particular port.

```
pascal ComponentResult AVEngineComponentGetFidelity
                                (ComponentInstance engineComponent,
                                 AVIDType portID,
                                 DMFidelityType* engineFidelity);
```

`engineComponent`
An instance of your engine component.

`portID` The ID of the port for which to return the fidelity measurement.

`engineFidelity` A number that is passed back to the caller to indicate how well the engine supports the port in question.

Supplementary Information

The Engine Component checks its own port ID to see if it matches. If it does, the component returns a fidelity measurement (`engineFidelity`) that indicates the engine should be used for this port ID, overriding any engines that are more generic.

AVEngineComponentTargetDevice

Your engine makes its connection to a particular port in response to a call to `AVEngineComponentTargetDevice`.

```
pascal ComponentResult AVEngineComponentTargetDevice
                                (ComponentInstance engineComponent,
                                 AVIDType portID);
```

`engineComponent` An instance of your engine component.

`portID` The ID of the port for which to return the fidelity measurement.

Supplementary Information

To find its port, the engine calls `DMNewDevicePortList(portID);`. The engine needs to talk to hardware, so it opens an instance of the Port Component. The Open selector of the port also opens an instance of the Device Component.

If the functionality for communicating with the hardware is stored in the Device Component, rather than the Port Component, the engine asks the port for the Device Component instance by calling `AVDevicePortGetDeviceComponent();`.

Registering Engine Components Multiple Times

There are special cases where you may need to register an Engine Component more than once. This occurs specifically if the Engine Component needs to share global data with other instances that might be open for the same device. For example, if the device is a display with a contrast control feature, it may take too long to read the contrast value from hardware each time you want to change contrast. In this case, the contrast value may be cached in the Engine Component.

The Manager Component registers the engine separately for each device, and stores the cached values in separate refCon globals for each registered engine. This means that the Manager Component, rather than the Panel Component, can call the engine's target selector. However, it can call it only once, at INIT time, to set the port ID for the engine. Later attempts to call the target selector will be ignored, since the engine is fixed on one device. When any call is made to the engine's `GetFidelity` selector, the engine can simply check its own port ID to see if it matches.

This is preferable to other solutions for a number of reasons:

- If the value is stored in the instance globals, other engines that are open for other panels will not have access to the updated value of the first engine and that engine's cache will become out of date without the engine realizing this.
- If the value is stored in the refCon globals, it conflicts with engines that are operating on different devices with different values.

Notification

When there are changes in the machine state, such as an increase or decrease in speaker volume, a change in contrast or brightness, and so forth, all components affected by the change are notified, provided they are registered with the Display Manager. The notification mechanism used in the AV Architecture allows these types of changes to be synchronized across multiple devices.

All components use the following notification messages. However, if you have a fully-featured suite of components (Panel, Engine, Port, and Device Components), it is the Engine Component that sends out the notification.

```
typedef pascal void (*DMExtendedNotificationProcPtr)(void*
userData,short theMessage,void* notifyData);

pascal OSErr
DMRegisterExtendedNotifyProc(DMExtendedNotificationProcPtr
notifyProc, void* notifyUserData,unsigned short
nofifyOnFlags,ProcessSerialNumberPtr whichPSN)

    = {0x303C,0x07EF,_DisplayDispatch};

pascal OSErr
DMRemoveExtendedNotifyProc(DMExtendedNotificationProcPtr
notifyProc,void* notifyUserData,ProcessSerialNumberPtr
whichPSN,unsigned short removeFlags)

    = {0x303C,0x0726,_DisplayDispatch};// Allows multiple
registrations to remove by userData, NOT just ProcPtr (which is
good for components).

pascal OSErr DMSendDependentNotification(ResType
notifyType,ResType notifyClass,DisplayIDType
portID,ComponentInstance notifyComponent)

    = {0x303C,0x0830,_DisplayDispatch};
```

This is not the only method of sending out a notification. "Notification Mechanisms" on page 58 provides additional information on this subject.

CHAPTER 4

Port Components

This chapter describes the Port Components that are part of the AV Architecture . The chapter explains why ports are a critical part of the AV Architecture; explains why Port Components are necessary, and describes how they work; describes the software functions associated with the Port Components.

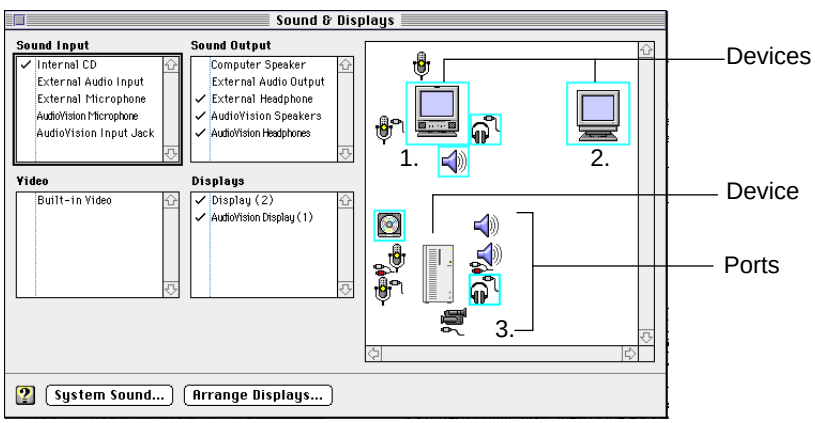
Why Have Ports?

Each AV device (computer, display, and so on) has ports that enable audio and video elements (such as displays, microphones, cameras, VCRs, and so on) to be connected to them.

With products, such as the Apple AudioVision display, that have many functions (brightness, contrast, speaker volume, headphone volume, microphone on/off, and so forth), it would be very confusing if all the panels associated with the functions appeared in one window at the same time.

The AV Architecture therefore partitions the control panels in a logical way that allows you to see groups of related panels in the same window. The most logical way to group the panels is by port, since each audio or video I/O function of an AV device has a discrete port associated with it. Figure 4-1 shows the Sound & Displays setup window. This window is typical of setups that group their panels by port. Three devices are displayed, as described below. Each device is surrounded by icons for the ports it supports.

Figure 4-1 Audio/Video Setup window



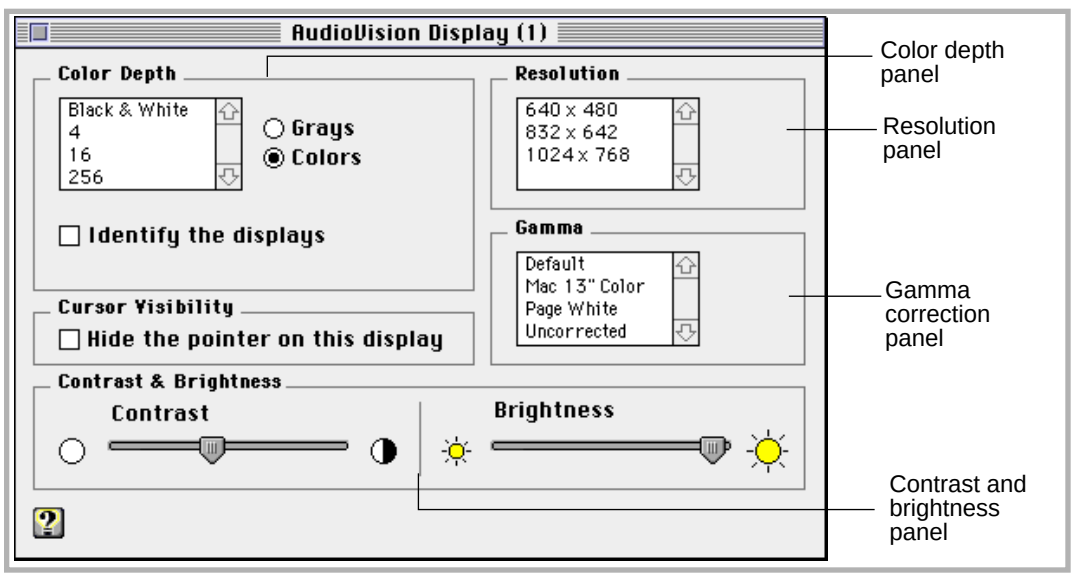
1. The first icon represents a display, such as an AudioVision display. The display has an integral microphone and speaker; ports for external speakers or headphone; and a port for a sound input, such as a microphone or CD player.

2. The second icon represents a regular display or monitor. This device has no external ports, but it does have intrinsic features that can be changed, for example brightness, contrast, resolution, and color depth.
3. The third icon represents a Macintosh computer. It has an internal CD port, internal speakers, ports for an external microphone or other sound input, a port for external speakers or headphones, and a port for a digital camera.

The lists on the left side of the window indicate which displays are connected to the system. In the example shown in Figure 4-2, there are two available displays: Display (2), and AudioVision Display (2). The other lists show the audio/video I/O ports associated with the devices shown on the right. The listings change according to the I/O devices connected to your system, and whether or not the ones present are functioning. For example, if you have a computer with an audio CD drive, Audio CD will not appear in the list if there is no CD in the drive.

When you select one of the devices, an associated panel is displayed on the screen. For example, if you select the AudioVision (1) display, you will see the panel shown in Figure 4-2. This panel allows you to change color depth, resolution, gamma correction, and brightness and contrast. It also allows you to identify the displays in your configuration, and to hide the pointer (cursor) on the display. The query button (?) allows you to access on-line help.

Figure 4-2 Setup control panel for an AudioVision 14 display



Each port is identified by an AV ID (audio/video identification) requested from the Display Manager at boot time. When an AV application wishes to show all applicable panels for a specific port, it asks the Display Manager for a list of panels. The Display Manager in turn asks all the Panel Components if they work on the selected port by calling the `AVPanelGetFidelity` function and passing in the AV ID of the port in question.

Why Have Port Components?

Each port has an associated Port Component that contains calls that return the name and image of the port, turn the port on and off, and provide hardware access to the port.

Storing Names and Icons

The Port Component stores the name string and icons that represent the port. It implements two functions, `AVPortGetName`, which returns the name of the port, and `AVPortGetGraphicInfo`, which returns the icon that is displayed to show the port on the user interface.

Turning the Port On and Off

In some cases, it is necessary simply to turn the port on or off. The meaning of *on* and *off* varies depending on the port. However, most ports have this option, and can implement it without actually opening a Panel Component or an Engine Component.

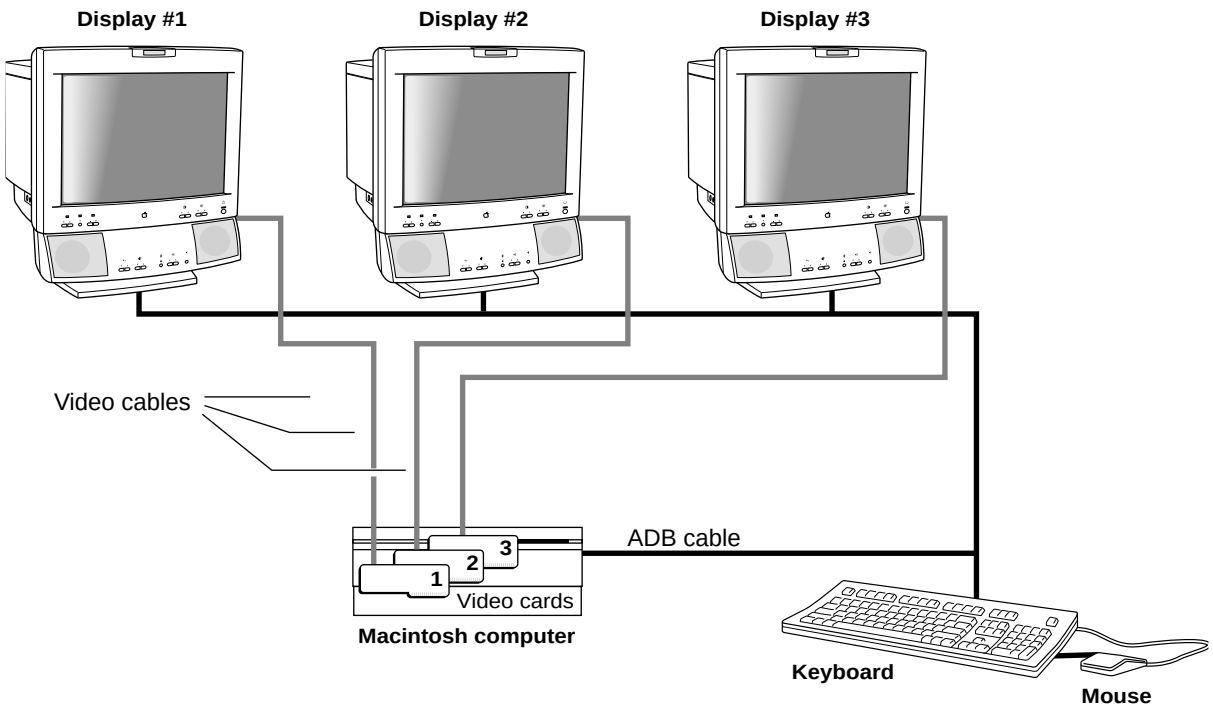
The `AVPortSetActive` and `AVPortGetActive` functions are used for this purpose. The caller calls the port's `AVPortSetActive` function to turn the port on or off, and then calls the `AVPortGetActive` function to check whether or not current is being supplied to the Port Component. The port does this by communicating directly with the hardware or with the Device Component.

Implementing Wiggling

Wiggling is important for hardware that has ports with more than one control path. For example, a computer may be connected to several AV displays, as shown in Figure 4-3. The computer has a video card for each display and is connected to each display by means of individual video cables. In addition, the computer is connected to each display, in daisy-chain fashion, by the Apple Desktop Bus (ADB) cable, and each display has a unique ADB address. Both the video cable and the ADB cable are used to control what appears on the display.

For example, if you are using an AppleVision display, when you press the Contrast Up button on the bezel of the display, the display sends a signal to the computer through the ADB. The computer then displays a pop-up window on the screen of the display on which the button was pressed. When you have more than one display connected, it is important to identify the display (device) associated with the ADB address, otherwise, the pop-up window may appear on the wrong screen.

Figure 4-3 Ports with more than one control path



Port Components have wiggle functions, `AVPortGetWiggle` and `AVPortSetWiggle`, that allow the correspondence to be made between each ADB address and the gDevice. These functions are generally used when multiple displays are present, but they may also be used for any ports that have more than one communications/control path.

Note
Wiggling is a process similar to toggling. The level of the sense line (Sense Line 2) in the video cable is changed, from 0 to 1, or from 1 to 0, to alert the computer that a given display is present. Wiggling is implemented only on display ports ◆

When wiggling is implemented, the entity that actually performs the pairing operation must be identified. For video/display ports, the Display Manager performs this function immediately after you register the port with the Display Manager. It does so by calling `DMNewAVIDbyPortComponent`. If the `portKind` is `AVVideoDisplayPortKind`, the Display Manager calls the `AVPortSetWiggle` function of the Port Component, and returns the value `true`. The Port Component then instructs the display, via the ADB (or other serial control path) to change Sense Line 2 to an abnormal state, meaning if it was 0 to change it to 1, and if it was 1 to change it to 0. In other words, it instructs the display to wiggle the line.

The Display Manager then queries each video driver to see which one detects this change. When it finds the appropriate driver, the pairing is made, and the Display Manager calls the `AVPortSetWiggle` function, and returns the value `false`. The Port Component then returns Sense Line 2 to its normal state.

If your port does not implement wiggling, the Display Manager cannot tell whether the two communication paths are for a single display or for two AV devices. This means that the user will see two ports listed under the list of ports in such applications as Sound & Displays. For instance, in the example described above, two ports would appear: one display port for controlling graphics device functions, such as bit depth; and one AppleVision display port for controlling contrast, brightness, and other AppleVision functions. The user could select either of these ports to access all of the display's functions.

Note
In some Apple publications, the term ***tagging*** is used instead of *wiggling*. ◆

Interfacing Between Engine Components and Hardware

When the physical hardware is grouped by port, different circuitry operates the screen, the microphone, speakers, and so forth. In this case, the Port Component is a convenient way for the Engine Components to communicate with the hardware, using a port call. This use of Port Components is optional. However, it makes sense, especially when the Port Component has to communicate with the hardware anyway, in order to turn a port on or off, or to implement wiggling. The Port Component can in turn call the Device Component, as described in Chapter 5, or communicate directly with the hardware.

Port Component Description

Port Components have the following component description:

`componentType = 'avdp';`	Indicates whether the component is an audio or video panel.
`componentSubType = '????'`	Identifies the specific engine.
`componentManufacturer = '????';`	Identifies the manufacturer.
`componentFlags = cmpWantsRegisterMessage;`	Makes sure the component is suitable for the machine.

Port Component Functions

There are several types of Port Component functions, those that:

- store names and icons
- turn the port on and off
- retrieve information from the Port Component
- detect changes on ports and implement wiggling

Storing Names and Icons

The functions in this section store the name and the icon of each port.

AVPortGetName

This function allows the caller to get the name of the port component, for example, External Microphone, External Headphones, and so on. The name is returned in `portName`. All Port Components must support this call.

```
pascal ComponentResult AVPortGetName
                              (ComponentInstance portComponent,
                               StringPtr portName);
```

`portComponent`	An instance of your Port Component.
`portName`	The name of the port to be returned by the function.

Supplementary Information

There is no supplementary information about this function.

AVPortGetGraphicInfo

This function allows the caller to get graphical information about the Port Component, that is the icon (or image) of the selected port. This could be a picture of a microphone, headphones, camera, and so forth. This function also allows you to set the position of the port icon with respect to the device icon. All Port Components must support this call.

```
pascal ComponentResult AVPortGetGraphicInfo
            (ComponentInstance portComponent,PicHandle *thePict,
             Handle *theIconSuite, AVLocationPtr theLocation);
```

`portComponent`	An instance of your Port Component.
`PicHandle`	Standard data type.
`*thePict`	Address of the `PicHandle` data type.

`*theIconSuite`	Address of a `Handle`.
`AVLocationPtr`	Address of an AV location data structure.
`theLocation`	Indicates the position of the AV location data structure. This is the position of the port icon with respect to the device icon. This field is defined as follows: `theLocation->locationConstant=k` Refer to the following section and to Figure 4-4 for further information.

Supplementary Information

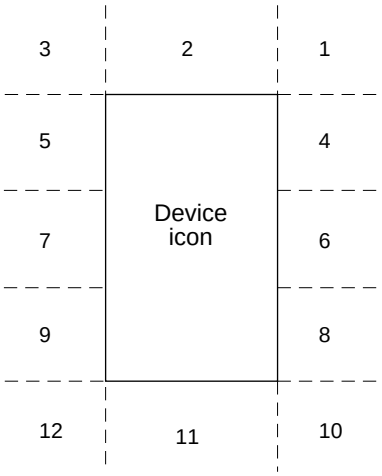
In response to this call, the Port Component should check the return parameters `thePict`, `theIconSuite`, and `theLocation`, to determine what information the caller is requesting.

If `thePict` is non nil (that is not 0), a handle to a picture of the port in PICT format should be returned in `*thePict`. If the picture is a resource that is retrieved via `GetResource`, `GetPicture`, and so on, you must be sure to detach the resource. If `theIconSuite` is non nil, a handle to an icon suite should be returned in `*theIconSuite`. If `theLocation` is non nil, the `AVLocationRec` that it points to should be filled out. The information that goes into the `AVLocationRec` is specific to each port and must be determined by each port.

The port icons are arranged round the device icons in a grid, as shown in Figure 4-4. The device may be a CPU, an AV display, a monitor, or any other piece of equipment with audio or video capabilitites. The grid has 12 different positions, and each port icon is assigned a position in the grid. The position is defined by the constant in the `theLocation` field, as shown in the listing below. These constants are in `AVComponents.h`.

```
/* Port Component Graphic Location Constants */
enum {
   kAVPortFirstPosition= 1,
   kAVPortSecondPosition,
   kAVPortThirdPosition,
   kAVPortFourthPosition,
   kAVPortFifthPosition,
   kAVPortSixthPosition,
   kAVPortSeventhPosition,
   kAVPortEighthPosition,
   kAVPortNinthPosition,
   kAVPortTenthPosition,
   kAVPortEleventhPosition,
   kAVPortTwelfthPosition,
};
```

Figure 4-4 Port location grid



Constants 1 through 12 represent port icon positions

Turning the Port On and Off

The functions in this category turn the related port on and off.

AVPortSetActive

This function allows clients to change the state of the port from active to inactive, or from inactive to active.

```
pascal ComponentResult AVPortSetActive
            (ComponentInstance portComponent,Boolean SetActive);
```

`portComponent` An instance of your Port Component.

`SetActive` Boolean that determines whether the port is active or inactive.

Supplementary Information

Ports may have inactive and inactive states, for example, the sound ports may be muted (inactive) or unmuted (active). `AVPortSetActive` provides the mechanism for toggling between states. If a port is called and `AVPortSetActive` is `true`, then the port is set to the active state. If `AVPortSetActive` is `false`, the port is set to the inactive state. If the port cannot be deactivated, it responds with an error message to this call.

AVPortGetActive

This function allows clients to query the state of the port and find out if it is active or inactive. All Port Components should support this call.

```
pascal ComponentResult AVPortGetActive
         (ComponentInstance portComponent, Boolean *isPortActive,
          Boolean *portCanBeActivated, Ptr Reserved);
```

`portComponent` An instance of your Port Component.

`*isPortActive` Boolean that indicates whether the port is active or inactive.

`*portCanBeActivated` Boolean that indicates whether or not the port can be activated.

Supplementary Information

If the port is active, `*isPortActive` should be set to `true`, otherwise, it should be set to `false`. If the port is the type that can be activated and deactivated, `*portCanBeActivated` should be set to `true`, otherwise, it should be set to `false`.

Retrieving Information From the Port Component

This section describes the two functions that enable you to retrieve information from Port Components.

AVDevicePortGetName

This function allows the caller to get the name of the associated port. A name must be returned. An empty string `()` is not an acceptable response.

```
pascal ComponentResult AVDevicePortGetName
            (ComponentInstance portComponent, Str255* portName);
```

`portComponent` An instance of your Port Component.

`portName` The name of the port to be returned by the function.

Supplementary Information

There is no supplementary information about this function.

AVDevicePortGetDeviceComponent

This function allows the caller to identify the Device Component that owns the port.

```
pascal ComponentResult AVDeviceGetPortDeviceComponent
                  (ComponentInstance portComponent,
                   CopmponentInstance* deviceComponentInstance);
```

`portComponent` An instance of your port component.

`deviceComponentInstance`
The port's associated device component instance.

Supplementary Information

There is no supplementary information about this function.

Detecting Changes on Ports

This section describes the Port Component functions that enable the Display Manager to detect how many displays have been connected to a device and to be aware of changes that may be made while the device is powered down or in a sleep state. Refer to "Implementing Wiggling" on page 42 for more information on this subject.

AVPortSetWiggle

This function sets the path code that allows the Display Manager to distinguish between different displays, video cards, and so on, present in the system.

```
pascal ComponentResult AVDevicePortSetWiggle
         (ComponentInstance portComponent, Boolean wiggleDevice);
```

`portComponent` An instance of your port component.

`wiggleDevice` Boolean representing the video port ID of the related display or video card.

Supplementary Information

Refer to "Implementing Wiggling" on page 42 for further information.

AVPortGetWiggle

This function allows the Display Manager get the path code set by `AVPortSetWiggle`.

```
pascal ComponentResult AVDevicePortGetWiggle
         (ComponentInstance portComponent, Boolean* wiggleDevice);
```

`portComponent` An instance of your port component.

`wiggleDevice` Boolean representing the video port ID of the related display or video card.

Supplementary Information

Refer to "Implementing Wiggling" on page 42 for further information.

CHAPTER 5

Device Components

This chapter describes the Device Components that are part of the AV Architecture . They are the lowest-level components in the architecture and represent an actual audio or video product. A device can be an intelligent display, a video card, a computer, and so forth. Device Components represent the physical box that contains all the ports. Refer to Chapter 4 for information on Port Components. This chapter explains why Device Components are a necessary part of the AV Architecture, and describes the software calls associated with Device Components.

Why Have Device Components?

Each device has an associated Device Component that contains the name and image of the device. The Device Component also provides access to the actual hardware layer, and provides storage space for information that is shared among the ports and engines.

Storing Names and Icons

The Device Component stores the name string and icons that represent the device. It implements two functions, `AVDeviceGetName`, which returns the name of the device, and `AVDeviceGetGraphicInfo`, which returns the icon that is displayed to show the device on the user interface.

Providing Interface With Hardware

It is sometimes convenient to have a common point through which all calls can be made to the hardware. This is particularly useful if all ports share the same communication path to the hardware and the same protocol. When it is providing this common point, the Device Component is acting as a kind of device driver. Both Engine Components and Port Components can communicate with the hardware through the Device Component. This is an optional use of the Device Component.

Providing Storage for Shared Information

Providing storage for shared information is another optional use of the Device Component. You may find it useful to cache hardware settings in the Device Component under any of the following conditions:

- When it is time-consuming to retrieve hardware settings directly from the hardware.
- When hardware settings are write only and cannot be read or checked.
- When the hardware functionalities of more than one engine or port are interrelated. For instance, if turning off a microphone port has the side effect of turning off the speaker port, then you might want to store flags for this side effect in the Device Component.

If the microphone panel and the speaker panel are displayed at the same time and you turn on the microphone by means of the microphone panel, the following sequence occurs:

- The microphone panel instructs its engine to turn on.
- The engine tells its Device Component to turn on the microphone, or alternatively issues the request to the Port Component, which in turn communicates with the Device Component.
- The Device Component changes its flags to show that the microphone is now on, and the speakers are now off.
- The engine sends out a Display Manager notification, indicating that something has changed.
- The speakers panel receives the notification and checks its engine to see if the speakers are on or off.
- The engine asks the Device Component if the speakers are on. Alternatively, the engine queries the Port Component, which in turn queries the Device Component.
- The Device Component checks its flags and reports that the speakers are off.

It is also possible to have a case where the functionality is not interrelated, but where two identical panels are displayed at the same time. The above process is used to make sure that the two panels remain synchronized with each other.

If you do not want to use this type of procedure, you may choose to register the engine separately and cache the engine settings in the refCon globals of the engine. You should be aware, though, that this is a limited solution. It is also more complicated, since the Manager Component must register and unregister the engines, depending on how many devices are present. The solution also fails to address the situation where the functioning of different engines on different ports is interrelated.

Device Component Functions

There are two Device Component functions that store the name and icon of the device. These functions are described below.

AVDeviceGetName

In response to the call, the Device Component returns the name of the device in the parameter `deviceName`. All Device Components must support this call.

```
AVDeviceGetName(ComponentInstance deviceComponent,
                StringPtr deviceName);
```

`deviceName` Name of the AV device, such as AudioVision, or AppleVision, that is being selected.

Supplementary Information

There is no supplementary information about this function.

AVDeviceGetGraphicInfo

This function allows the caller to retrieve graphic information about the Device Component. When a call is made to this function, it loads a `PicHandle` resource and returns the address of the `Handle`. It then loads an icon suite resource and returns its address in `theIconSuite`. All Device Components must support this call.

```
AVDeviceGetGraphicInfo  (ComponentInstance deviceComponent,
                         PicHandle *thePict,
                         Handle *theIconSuite,
                         AVLocationPtr theLocation);
```

`PicHandle`	Standard data type.
`thePict`	Address of a `PicHandle` data type.
`theIconSuite`	Address of a `Handle`.
`AVLocationPtr`	Address of an AV location data structure.
`theLocation`	Indicates the location of an AV location data structure. Currently it is always 0, as shown below: `theLocation->LocationConstant=0`

Supplementary Information

In response to this call, the Device Component should check the return parameters `thePict`, `theIconSuite`, and `theLocation`, to determine what information the caller is requesting.

If `thePict` is non nil (that is not 0), a handle to a picture of the device in PICT format should be returned in `*thePict`. If the picture is a resource that is retrieved via `GetResource`, `GetPicture`, and so on, you must be sure to detach the resource. If `theIconSuite` is non nil, a handle to an icon suite should be returned in `*theIconSuite`. If `theLocation` is non nil, the `AVLocationRec` that it points to should be filled out.

Currently, all devices have a nil (zero) location constant. In this case, they should be ignored and not filled out.

CHAPTER 6

More Information About the AV Architecture

This chapter contains supplementary information about the AV Architecture. It covers the following topics:

- storing data in components
- managing component storage
- the notification mechanism
- utility components
- library of utility functions

Strategies for Storing Data in Components

Components have two mechanisms for storing data: instance globals and common globals. Instance globals are private to a given instance of the component, and they are retrieved through the `Set/GetComponentInstanceStorage()` functions. Common globals apply to all instances of a given component.

Note
An *instance* is one particular representative within a group of components. ◆

If components are of the same type or subtype, any number of them can share a single set of common globals. A handle to this data structure is stored in the component's refCon, and it is accessed through the `Set/GetComponentRefCon` functions. Figure 6-1 shows the ownership model for two instances of an AV component, which might be a Panel, Engine, Port, or Device Component. Generally, an AV component always needs a set of instance globals and may never need to share common globals with other components.

Managing Component Storage

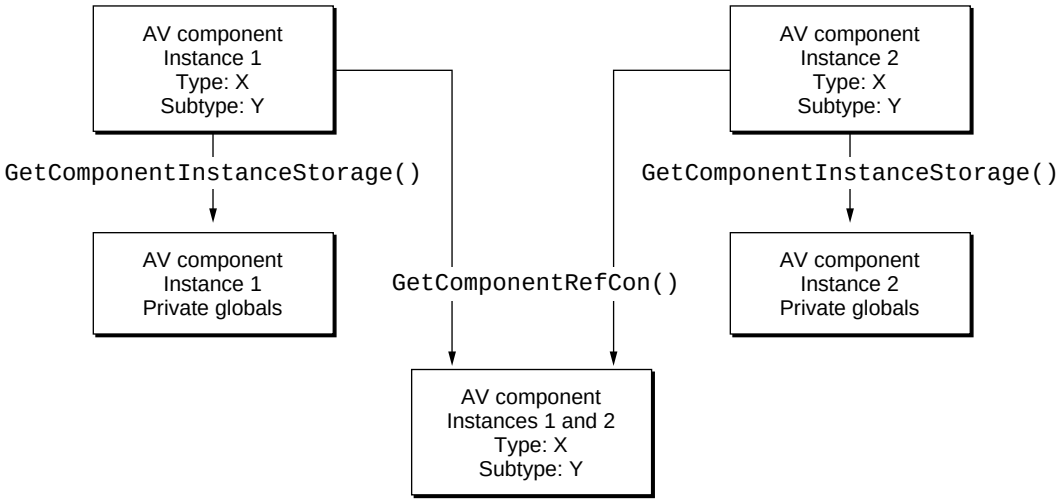
There are certain well-defined times when you may want to create or destroy the two types of storage provided by instance and common globals. You must be careful to create or destroy the appropriate storage, particularly when you are dealing with common globals.

Creating and Destroying Instance Globals

Instance globals are normally created in the `AVPanelOpen()` routine. "AVPanelSetCustomData" on page 23 provides detailed information about this routine. To create an instance global:

- Allocate a handle large enough to store the information you will need.
- Call `SetComponentInstanceStorage()`.

Figure 6-1 Ownership model



You can dispose of an instance global in the `AVPanelClose()` routine. "AVPanelClose" on page 21 provides detailed information about this routine. To dispose of an instance global:

- Dispose of any handles that are stored within the global. This is very critical.
- Dispose of the global's handle.

Creating and Destroying Common Globals

There are also well-defined times when common globals (if used) are allocated and disposed of. However, management of common globals is dependent on how many instances of a given component are open at the time you wish to allocate or dispose of the common global.

If you are sharing common globals among multiple instances, and you wish to allocate a common global, you may use the following process:

- In the `PanelOpen()` routine, count the number of open instances of the component.
- If there is only one open instance, it is safe to allocate a handle for the common globals, initialize it as needed, and then call `SetComponentRefCon()` with that handle.
- If there is more than one instance open, you need not call `SetComponentRefCon()`, because all instances will share the refCon, and will therefore share the common globals handle.

The process is similar for disposing of common globals:

- In the `PanelClose()` routine, count the number of instances of the component.
- If there is only one, then it is time to dispose of the common globals.

- Dispose of any handles contained in the globals, then dispose of the globals themselves.
- When you have disposed of the common globals, you can go on to dispose of the instance globals, as described in the previous section.

Notification Mechanisms

If you have a fully featured suite of components, consisting of Panel, Engine, Port, and Device Components, notification of changes in the machine state is generally provided by the Engine Component. "Notification" on page 38 provides information about this type of notification. However, there are other methods of sending notification, as described in the following sections.

Display Notification

The Display Manager provides a mechanism that allows any software entity to notify any other software entity that a change has been made in the system configuration. This mechanism is known as a display notification.

IMPORTANT
This is type of notification is not the same as that provided by the Toolbox Notification Manager. ▲

The AV Architecture uses display event notification to keep all open components synchronized with what is happening. For example, if multiple instances of the Bit Depth panel are open, and you make a `SetDepth()` Toolbox call, then all the open Bit Depth panels will update themselves to reflect the current bit depth and the color state of the screen that was changed.

The AV Architecture is able to keep track of these types of changes because the open Bit Depth panels are registered with the Display Manager and are therefore notified whenever there is a change in the graphics environment. If you have a variety of AV components that need to track with each other, you can use the AV notification mechanism. To do this, components should register themselves with a call to `DMRegisterExtendedNotifyProc()`, usually from within the `PanelInstall()` function.

Whenever there is a change in the graphics environment, the Display Manager automatically broadcasts an event using the `kDMNotifyEvent` message. The types of changes include

- bit depth, implemented through `SetDepth()`
- resolution, implemented through `DMSetDisplayMode()`
- gDevice position, implemented through `DMMoveDisplay()`
- mirroring, implemented through `DMMirrorDevices()`

- adding displays, implemented through `DMAddDisplay()`
- removing displays, implemented through `DMRemoveDisplay()`

One of the parameters to the `DMRegisterExtendedNotifyProc()` function is the address of a notification handler routine. Whenever the Display Manager broadcasts a notification event, this handler routine is called.

Resolution Panel Notification

The Resolution Panel uses the Display Manager as its hardware abstraction. When this panel asks for a resolution change, the Display Manager panel automatically sends out a notification.

Cursor Visibility Panel Notification

The Cursor Visibility Panel does not use Engine, Port, or Device Components. In addition, the Display Manager knows nothing about this panel. When a change is made in the panel, the panel itself makes a couple of new Toolbox calls, and then explicitly sends out the notification.

Utility Components

Utility components handle service requests and have the tools needed to respond to such requests from a component client. Currently, there is only one utility component. This is the Preferences Component described below.

Preferences Component

When an AV component needs to save information on disk, and you choose not to save it in the standard AV preferences file, you can save it in your own preferences file using the Preferences Component.

Your Component (Panel, Engine, Device, Port) can open an instance of the Preferences Component and save a reference to it in the panel's instance globals. "Strategies for Storing Data in Components" on page 56 provides further information on this subject. When the panel is being closed, you can use this component to create a private file and store your data there. Later, when the panel is being opened, it can open an instance of the Preferences Component and ask it to retrieve the information that was previously stored. Your panel can then set up its internal data using this information.

Library of Utility Functions

Utility functions are tools provided by Apple for developers as part of the software development kit. The AV library of utility functions contains common pieces of functionality that may be useful to developers. Currently, they are utilities associated with getting globals from a panel, and with getting and setting the application's resFile.

Getting Globals

This utility enables you to get the globals from a panel within a dialog box filter function. Since AV panels create their human interface pieces using dialog box items and dialog box windows, anything that is not a standard item, such as a radio button, checkbox, static text, and so on, must have its drawing and updating managed by the panel itself. This means that you must write a custom dialog box item draw procedure, which has a fixed API. This API gives only the dialog box window pointer and the DITL index of the item that needs to be drawn. The assumption is that you will use the refCon of the dialog box window to store a handle to your private information. You can then get the information needed to draw or update your custom item using this handle.

Because the AV panel window uses the refCon for storing a wide variety of information that is not necessarily related to any given panel in the window, you do not want to open up the storage format and expose the data. For this reason, you should use the following utility routine provided in the AV library of utility functions:

```
Handle AVGetPanelGlobalsFromPanelWindow()
```

This function requires only the dialog box window pointer and the absolute item index as input for any item in the panel. The function knows how to traverse the information in the panel window storage handle, and it will return a handle to the globals for the panel. The caller must not dispose of this handle. It will be taken care of when the panel is closed.

IMPORTANT
The absolute item index is required, not the relative item index. ▲

Getting and Setting the resFile

This utility enables you to get the application's resource file (resFile) from an arbitrary component, or to set the application's resFile in an arbitrary component. When a component needs to access any of its resources, it must first make sure that it sets the current resource file to its own resFile, using the `SetComponentResFile()` function.

Because the component is likely to be called from some kind of controlling application, it is important to save and restore whatever resource file was current before the component changed files. It is not sufficient merely to call `CurResFile()` to find out what the current file is before switching to the component's file.

For this reason, the AV architecture provides the following routine:

```
short AVGetApplicationResFile()
```

Your panel can either save the value `()` in its private globals, or it can call the function each time it needs this information. Whenever your panel sets its component resource file, you should make a call to `UseResFile(appResFile)` before leaving the component function.

Manager Component

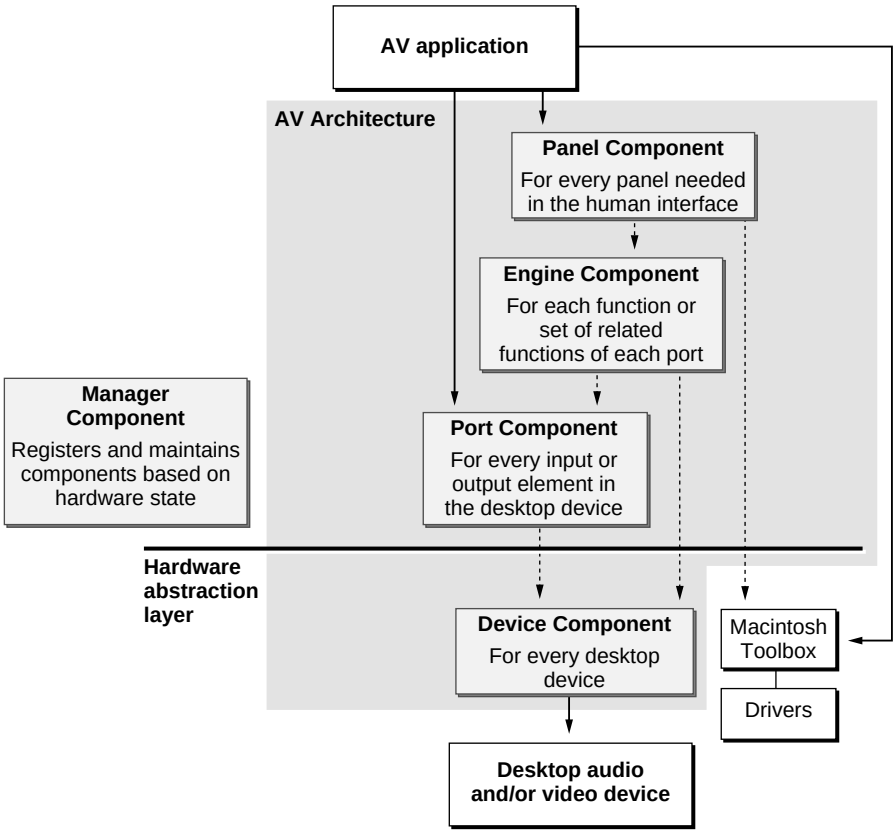
The Manager Component is not a required part of the AV Architecture. However, it is often convenient and, in certain cases essential, to define one Manager Component that registers and unregisters all your other components. The Manager Component is needed if your components must load in a certain order, or if it is possible to remove or add essential hardware while the computer, for example a PowerBook, is asleep. This chapter explains the key functions of the Manager Component.

The Manager Component is responsible for:

- Loading all the other components.
- Registering Device and Port Components with the Display Manager and establishing connections between Device and Port Components.
- Registering and unregistering components when the system is initialized, shuts down, goes into sleep mode, wakes from sleep mode.
- Answering queries about the status of the component group as a whole.

Figure A-1 shows the relationship between the Manager Component and the AV Architecture components.

Figure A-1 Manager Component



Loading Components

The Manager Component shields your components from the Component Manager, making sure that the Component Manager does not load them in the wrong order. It is also responsible for loading the components in the correct order.

The Manager Component is the only component that has a ‘thng’ resource, and is therefore the only component automatically loaded by the Component Manager at boot time. All other components should have ‘gnht’ resources. The Manager Component performs in a similar way to the Loader Component, the only difference being that no ‘thld’ resource is used, since registration may vary depending on the state of the hardware present.

Typically you will ship one extension file for your product. This extension file contains the Panel, Engine, Port, and Device Components as ‘gnht’ resources, rather than ‘thng’ resources. This means the Component Manager will not try to register them. The

Manager Component is stored in the extensions as a regular ‘thng’ resource, and when it is registered, it in turn finds the ‘gnht’ resources for the device, ports, and engines, and registers them according to the hardware that is present.

Registering Components

The Manager Component should be able to determine how many devices are present. If there is more than one hardware device attached to the Macintosh, it is preferable to register the Device and Port Components multiple times, once for each like device attached to the system. This means that there will be separate refcon globals for device variables. You may question whether it is better to create multiple instances of the Device Component or to register separately. Generally you should register separately.

Component vs. INIT

In most case you should use a component rather than an INIT. This is because systems earlier than System 7.1 need QuickTime INIT to install the Component Manage. Therefore, if you use an INIT, you run the risk of the name of the INIT starting with a letter earlier in the alphabet than Q (QuickTime). In which case, your components would not be registered, since the Component Manager would not have been installed.

There is one case when you must use an INIT, and take a chance with systems earlier than System 7.1. This is when you need a ‘sysz’ resource, such as the following example, to temporarily allocate more heap space for your components during boot time.

Example:

```
data ‘sysz’ (0) {
     $”0002 0000”
};
```

However, ‘sysz’ resources only take effect when you have an ‘INIT’ in the same files as the ‘sysz’ resource. You should therefore simply load the Manager Component in a very small INIT. When you do this, make sure you change the Manager Component’s ‘thng’ resource to a ‘gnht’ resource, to hide it from the Component Manager. Otherwise, it will load twice.

Source code sample:

```
#include <Components.h>

void MyINIT()
{
   Component theCompID;
   ComponentResourceHandle componentResHdl;
   componentResHdl = (ComponentResourceHandle)
```

```
Get1Resource('gnht',
      kManagerComponentResourceID);
   if (componentResHdl)
      theCompID = RegisterComponentResource (componentResHdl,

         registerComponentGlobal);
   return;
}
```

Interaction With Device and Port Components

The Manager Component provides a convenient way of registering Device and Port Components with the Display Manager and of establishing connections between Device and Port Components.

All Device and Port Components must be registered with the Display Manager. You can do this using the `DMNewAVIDByDeviceComponent` and `DMNewAVIDByPortComponent` calls. When these calls return new AV IDs, you let your Device and Port Components knows what their own AV IDs are by making the standard `AVDeviceSetAVID` and `AVPortSetAVID` calls.

In addition, you must establish connections between the Device and Port Components. Port Components should know to which Device Component they are attached. You can do this using the port call `AVPortSetDeviceAVID`.

Cleaning Up During Sleep or Shutdown

The PowerBook computer goes into sleep mode during periods of inactivity. During this time you might add or remove critical devices. For example you might add or remove a display. Similarly, devices may be removed or connected while the PowerBook is shutdown and then be present when it is powered up again.

The Manager Component manages the registration and unregistration of components during these periods. The following points are critical:

- You should not register a component when the related hardware is not available.
- If the hardware your components use changes while the PowerBook is asleep, you must make sure that they are registered (if added), or unregistered (if removed) when the PowerBook wakes from sleep mode.
- You should perform some kind of initialization when the PowerBook wakes from sleep mode.
- At shutdown you should clean up and remove unwanted components.

Manager Component

The Manager Component receives sleep, wakeup, and shutdown notifications, and goes through the processes necessary to handle these situations. The source code required to accomplish this is shown in the following sample.

Source Code Sample

```
#include <Power.h>

typedef struct SleepInfoRec
{
   SleepQRec mySleepQRec;
   long mySlpRefCon;
} SleepInfoRec, *SleepInfoRecPtr, **SleepInfoRecHandle;

void MySleepProc(void);

{ // install a sleep task
   OSErr result = noErr;
   long pmgrAttributes;

      result = Gestalt(gestaltPowerMgrAttr, &pmgrAttributes);
   if (result == noErr && (pmgrAttributes & (1<<gestaltPMgrExists)))
   { // Power Manager is present
      SleepInfoRecPtr sleepInfoRecPtr = NULL;
      Handle mySleepProcHandle = NULL;

         result = LoadPieceOfCode(kMyCodeResourceType,
            kMySleepResID,
            &mySleepProcHandle);
      if (result == noErr)
      {
         sleepInfoRecPtr = (SleepInfoRecPtr) NewPtrSysClear(
            sizeof(SleepInfoRec));
         if (sleepInfoRecPtr != NULL)
         {
            sleepInfoRecPtr->mySleepQRec.sleepQLink = NULL;
            sleepInfoRecPtr->mySleepQRec.sleepQType = slpQType;
            sleepInfoRecPtr->mySleepQRec.sleepQProc = (SleepQUPP)
*mySleepProcHandle;
            sleepInfoRecPtr->mySleepQRec.sleepQFlags = 0;
            sleepInfoRecPtr->mySlpRefCon = 0; // could use this for something
            SleepQInstall((SleepQRecPtr)sleepInfoRecPtr);
         }
      }
   }
}
```

```
{ // install a shutdown task
   Handle myShutdownProcHandle = NULL;

   result = LoadPieceOfCode(kMyCodeResourceType, kMyShutdownResID,
         &myShutdownProcHandle);
   if (result == noErr)
      ShutDwnInstall((ShutDwnUPP) *myShutdownProcHandle, (sdOnDrivers));
}

OSErr LoadPieceOfCode(ResType theType, short theID, Handle *theProcHandle)
{
   OSErr result = noErr;
   Handle myProcHandle = NULL;
   long myProcSize;

   *theProcHandle = NULL;

   // first get the resource without actually loading it
   SetResLoad(false);
   myProcHandle = GetResource(theType, theID);
   SetResLoad(true);

   if (myProcHandle != NULL)
   {
      // see how big it is
      myProcSize = GetResourceSizeOnDisk(myProcHandle);
      result = ResError();
      if (result == noErr)
      {
         // reserve this much memory at the bottom of the System heap
         ReserveMemSys(myProcSize);
         LoadResource(myProcHandle);
         result = ResError();
         if (result == noErr)
         {
            DetachResource(myProcHandle);
            HLock(myProcHandle);
            *theProcHandle = myProcHandle;
         }
      }
   }
   else result = -1;

      return result;
}
```

The sleep procedure is loaded in the previous code sample as a separate code resource, and it may look like the following code sample.

Source Code Sample

```
#include <Components.h>

extern void MyAllowSleepRequest(void)
   ONEWORDINLINE(0x7000); // MOVEQ #0, D0

extern void MyDenySleepRequest(void)
   ONEWORDINLINE(0x7001); // MOVEQ #1, D0

extern SleepInfoRecPtr MyGetSleepInfoPtr(void)
   ONEWORDINLINE(0x2E88); // MOVE.L A0, (A7)

extern long MyGetSleepCommand(void)
   ONEWORDINLINE(0x2E80); // MOVE.L D0, (A7)

void MySleepProc()
{

   SleepInfoRecPtrmySleepInfoPtr;
   long              mySleepCommand;

   mySleepInfoPtr = MyGetSleepInfoPtr();
   mySleepCommand = MyGetSleepCommand();

      switch (mySleepCommand)
   {
      case sleepRequest:

         MyAllowSleepRequest();
         break;
      case sleepDemand:
      case sleepWakeUp:
      {
         ComponentDescription compDesc;
         Component managerCompID = NULL;

         compDesc.componentType = kManagerType;
         compDesc.componentSubType = kManagerSubType;
         compDesc.componentManufacturer = kManufacturerType;
         compDesc.componentFlags = 0;
         compDesc.componentFlagsMask = kAnyComponentFlagsMask;

         managerCompID = FindNextComponent(managerCompID, &compDesc);
```

```
            if (managerCompID != NULL)
            {
               ComponentInstance managerInstance;

               managerInstance = OpenComponent(managerCompID);

               if (mySleepCommand == sleepDemand)
                  ManagerComponentSleep(managerInstance);
               else
                  ManagerComponentWake(managerInstance);

                  CloseComponent(managerInstance);
            }
         }
            break;
         case sleepRevoke:
         default:
            break;
      }
}
```

You can define the `ManagerComponentSleep` and `ManagerComponentWake` selectors in the Manager Component so that they register and unregister your other components as needed to match the hardware that is present. The shutdown procedure can be even simpler. You can just find the Manager Component and call a `ManagerComponentShutdown` selector that you have defined.

Reporting on the State of the Component Group

The Manager Component can also serve as a representative of your software as a whole. If you have a custom AV application that makes heavy use of the AV components, you may decide that it should not run at all if your components are not loaded correctly.

Your application should first check for the existence of the Manager Component, using the `FindNextComponen`t call. If the Manager Component cannot be found, the components are definitely not loaded.

If the Manager Component is found, then your application, using `MyManagerComponentGetState()`, can request that it check the state of the component set. If all your components were not loaded at boot time, `MyManagerComponentGetState()` can return an error or a constant to indicate that your application should not run.

Glossary

Apple event An Apple event is a high-level event that adheres to the Apple Event Interprocess Messaging Protocol. An Apple event consists of attributes, including the event class and event ID that identify the event and its task. It also contains parameters that contain data used only by the target application. You will find detailed information about Apple events in *Inside Macintosh, Volume VI*.

architecture A set of design principles that specifies the relationship among components, in this context, Panel, Engine, Port, and Device Components.

DITL The DITL (dialog item list) is a list of resources associated with a given control panel.

hardware abstraction This is a process that takes hardware functionality and gives it a name, thus concealing the hardware implementation from the software. The hardware abstraction layer acts as a liaison between the software element and the hardware element.

Virtual Photometry Technology (VPT) A proprietary Apple technique used in imaging devices, such as video displays, to ensure accurate color on the screen.

Index

A, B

C

D

E

F

G

H

I, J, K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Select 360 printer. Final pages were created on the Apple LaserWriter Pro 630. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITER
Joyce D. Mann

DEVELOPMENTAL EDITOR
John Hammett

ILLUSTRATOR
Sandee Karr

Special thanks to
Aaron Ludtke, Mark Taylor,
Greg Mullins, and William Sheet