Component Manager

Contents

Introduction to Components 6-3
About the Component Manager 6-4
Using the Component Manager 6-6
Opening Connections to Components 6-7
Opening a Connection to a Default Component 6-7
Finding a Specific Component 6-8
Opening a Connection to a Specific Component 6-9
Getting Information About a Component 6-10
Using a Component 6-11
Closing a Connection to a Component 6-12
Creating Components 6-13
The Structure of a Component 6-13
Handling Requests for Service 6-18
Responding to the Open Request 6-19
Responding to the Close Request 6-21
Responding to the Can Do Request 6-22
Responding to the Version Request 6-22
Responding to the Register Request 6-23
Responding to the Unregister Request 6-24
Responding to the Target Request 6-25
Responding to Component-Specific Requests 6-26
Reporting an Error Code 6-28
Defining a Component's Interfaces 6-28
Managing Components 6-30
Registering a Component 6-30
Creating a Component Resource 6-32
Establishing and Managing Connections 6-34
Component Manager Reference 6-37
Data Structures for Applications 6-37
The Component Description Record 6-37

Component Identifiers and Component Instances 6-40
Routines for Applications 6-41
Finding Components 6-42
Opening and Closing Components 6-44
Getting Information About Components 6-47
Retrieving Component Errors 6-51
Data Structures for Components 6-52
The Component Description Record 6-52
The Component Parameters Record 6-54
Routines for Components 6-56
Registering Components 6-57
Dispatching to Component Routines 6-63
Managing Component Connections 6-65
Setting Component Errors 6-69
Working With Component Reference Constants 6-70
Accessing a Component's Resource File 6-71
Calling Other Components 6-73
Capturing Components 6-75
Targeting a Component Instance 6-77
Changing the Default Search Order 6-78
Application-Defined Routine 6-79
Resources 6-80
The Component Resource 6-80
Summary of the Component Manager 6-86
Pascal Summary 6-86
Constants 6-86
Data Types 6-87
Routines for Applications 6-89
Routines for Components 6-90
Application-Defined Routine 6-92
C Summary 6-92
Constants 6-92
Data Structures 6-93
Routines for Applications 6-95
Routines for Components 6-96
Application-Defined Routine 6-97
Assembly-Language Summary 6-98
Trap Macros 6-98
Result Codes 6-99

This chapter describes how you can use the Component Manager to allow your application to find and utilize various software objects (components) at run time. It also discusses how you can create your own components and how you can use the Component Manager to help manage your components. You should read this chapter if you are developing an application that uses components or if you plan to develop your own components.

The rest of this chapter

- contains a general introduction to components and the features provided by the Component Manager
- discusses how to use the facilities of the Component Manager to call components
- describes how to create a component

Several of the sections in this chapter are divided into two main topics: one describes how applications can use components, and one describes how to create your own components. If you are developing an application that uses components, you should focus on the material that describes how to use existing components—you do not need to read the material that describes how to create a component. If you are developing a component, however, you should be familiar with all the information in this chapter.

For information on a specific component, see the documentation supplied with that component. For example, for information on the components that Apple supplies with QuickTime, see *Inside Macintosh: QuickTime Components*.

Introduction to Components

A **component** is a piece of code that provides a defined set of services to one or more clients. Applications, system extensions, as well as other components can use the services of a component. A component typically provides a specific type of service to its clients. For example, a component might provide image compression or image decompression capabilities; an application could call such a component, providing the image to compress, and the component could perform the desired operation and return the compressed image to the application.

Multiple components can provide the same type of service. For example, separate components might exist that can compress an image by 20 percent, 40 percent, or 50 percent, with varying degrees of fidelity. All components of the same type must support the same basic interface. This allows your application to use the same interface for any given type of component and get the same type of service, yet allows your application to obtain different levels of service.

The Component Manager provides access to components and manages them by, for example, keeping track of the currently available components and routing requests to the appropriate component.

The Component Manager classifies components by three main criteria: the type of service provided, the level of service provided, and the component manufacturer. The Component Manager uses a **component type** to identify the type of service provided by a component. Like resource types, a component type is a sequence of four characters. All components of the same component type provide the same type of services and support a common application interface. For example, all image compressor components have a component type of `'imco'`. Other types of components include video digitizers, timing sources, movie controllers, and sequence capturers.

Note

Component types consisting of only lowercase characters are reserved for definition by Apple. You can define component types using other combinations of characters, but you must register any new component types with Apple's Component Registry Group (AppleLink REGISTRY). ◆

The Component Manager allows components to identify variations on the basic interface they must support by specifying a four-character **component subtype.** The value of the component subtype is meaningful only in the context of a given component type. For example, image compressor components use the component subtype to specify the compression algorithm supported by the component.

All components of a given type-subtype combination must support a common application interface. However, components that share a type-subtype specification may support routines that are not part of the basic interface defined for their type. In this manner, components can provide enhanced services to client applications while still supporting the basic application interface.

Finally, the Component Manager allows components to have a four-character manufacturer code that identifies the manufacturer of the component. You must register your component with Apple's Component Registry Group to receive a manufacturer code for your component. The manufacturer code allows applications to further distinguish between components of the same type-subtype.

About the Component Manager

The Component Manager provides services that allow applications to obtain run-time location of and access to functional objects (in much the same way that the Resource Manager allows applications that are running to access data objects dynamically).

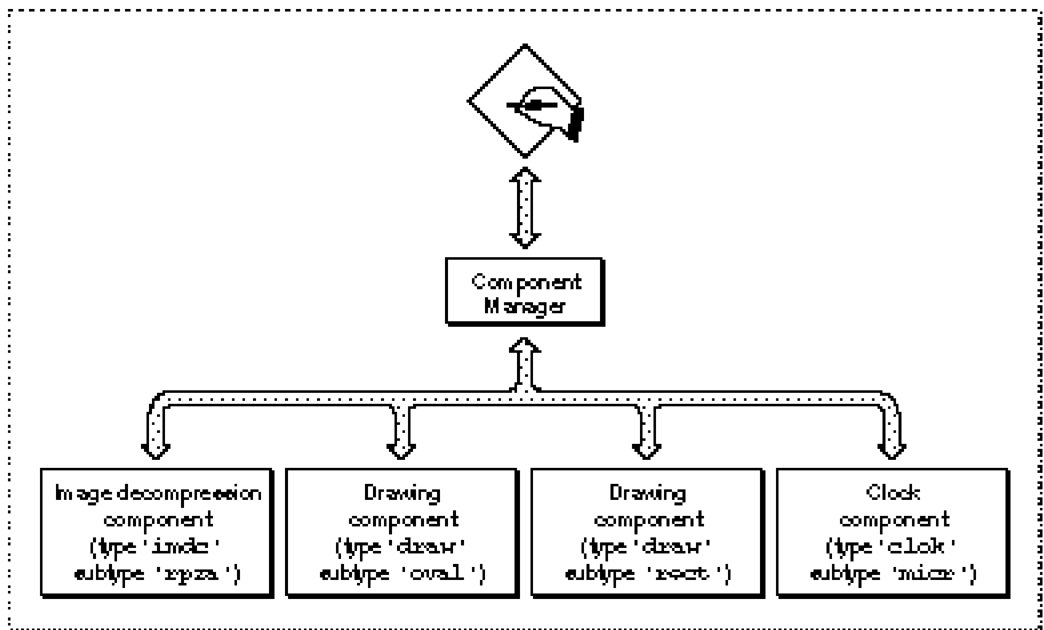
The Component Manager creates an interface between components and clients, which can be applications, other components, system extensions, and so on. Instead of implementing support for a particular data format, protocol, or model of a device, you can use a standard interface through which your application communicates with all components of a given type. You can then use the Component Manager to locate and communicate with components of that type. Those components, in turn, provide the appropriate services to your client application.

Given a particular component type, the Component Manager can locate and query all components of that type. You can find out how many components of a specific type are available and you can get further details about a component's capabilities without having to open it first. For each component, the Component Manager keeps track of many characteristics, including its name, icon, and information string.

For example, components of type `'imdc'` provide image decompression services. All components of type `'imdc'` share a common application interface, but each image decompressor component may support a unique compression technique or take advantage of a special hardware implementation. Individual components may support additions to the defined application interface, as long as they support the common routines. Any algorithm-dependent or implementation-dependent variations of the general decompression interface can be implemented by each `'imdc'` component as extensions to the basic interface.

Figure 6-1 shows the relationship between an application, the Component Manager, and several components. Applications and other clients use the Component Manager to access components. In this figure, four components are available to the application: an image decompression component (of type `'imdc'`), two drawing components (of type `'draw'`), and a clock component (of type `'clok'`). Note that the two drawing components have different subtypes: `'oval'` and `'rect'`. The drawing component with subtype `'oval'` draws ovals, and the drawing component with subtype `'rect'` draws rectangles.

Figure 6-1 The relationship between an application, the Component Manager, and components



The Component Manager allows a single component to serve multiple client applications at the same time. Each client application has a unique access path to the component. These access paths are called **component connections.** You identify a component connection by specifying a **component instance.** The Component Manager provides this component instance to your application when you open a connection to a component. The component maintains separate status information for each open connection.

For example, multiple applications might each open a connection to an image decompression component. The Component Manager routes each application request to the component instance for that connection. Because a component can maintain separate storage for each connection, application requests do not interfere with each other and each application has full access to the services provided by the component. (See Figure 6-2 on page 6-34 for an illustration of multiple aplications using the services of the same component.)

Using the Component Manager

This section describes how you can use the Component Manager to

- gain access to components
- locate components and take advantage of their services
- get information about a component
- close a connection to a component

The Component Manager is available in System 7.1 or later and may be present in System 7. To determine whether the Component Manager is available, call the `Gestalt` function with the `gestaltComponentMgr` selector and check the value of the `response` parameter.

```
CONST
   gestaltComponentMgr      = 'cpnt';
```

The `Gestalt` function returns in the `response` parameter a 32-bit value indicating the version of the Component Manager that is installed. Version 3 and above supports automatic version control, the unregister request, and icon families. You should test the version number before using any of these features.

This section presents several examples demonstrating how to use components and the Component Manager. All of these examples use the services of a drawing component—a simple component that draws an object of a particular shape on the screen. Drawing components have a component type of `'draw'`. The component subtype value indicates the type of object the component draws. For example, a drawing component that draws

an oval has a component subtype of 'oval'. For information on creating your own components and for listings that show the code for a drawing component, see "Creating Components" beginning on page 6-13.

Opening Connections to Components

When your application requires the services of a component, you typically perform these steps:

- open a connection to the desired component
- use the services of the component
- close the connection to the component

The following sections describe each of these steps in more detail.

Opening a Connection to a Default Component

Your application must use the Component Manager to gain access to a component. The first step is to locate an appropriate component. You can locate the component yourself, or you can allow the Component Manager to locate a suitable component for you. Your application then opens a connection to that component. Once you have opened a connection to a component, you can use the services provided by that component. When you have finished using the component, you should close the connection.

If you are interested only in using a component of a particular type-subtype and you do not need to specify any other characteristics of the component, use the `OpenDefaultComponent` function and specify only the component type and subtype—the Component Manager then selects a component for you and opens a connection to that component. This is the easiest technique for opening a component connection. The `OpenDefaultComponent` function searches its list of available components and attempts to open a connection to a component with the specified type and subtype. If more than one component of the specified type and subtype is available, `OpenDefaultComponent` selects the first one in the list. If successful, the `OpenDefaultComponent` function returns a component instance that identifies your connection to the component. You can then use that connection to employ the services of the selected component.

This code demonstrates the use of the `OpenDefaultComponent` function. The code opens a connection to a component of type 'draw' and subtype 'oval'—a drawing component that draws an oval.

```
VAR
   aDrawOvalComp: ComponentInstance;

   aDrawOvalComp := OpenDefaultComponent('draw', 'oval');
```

If it cannot find or open a component of the specified type-subtype, the `OpenDefaultComponent` function returns a function result of `NIL`.

To open a connection to a component with a specific type-subtype-manufacturer code or with other specified characteristics, first use the `FindNextComponent` function to find the desired component, then open the component using the `OpenComponent` function. These operations are described in the next two sections.

Finding a Specific Component

If you are interested in asserting greater control over the selection of a component, you can use the Component Manager to find a component that provides a specified service. For example, you can use the `FindNextComponent` function in a loop to retrieve information about all the components that are registered on a given computer. Each time you call this function, the Component Manager returns information about a single component. You can obtain a count of all the components on a given computer by calling the `CountComponents` function. Both of these functions allow you to specify search criteria, for example, by component type and subtype, or by manufacturer. By using these criteria to narrow your search, you can quickly and easily find a component that meets your needs.

You specify the search criteria for the component using a component description record. A component description record is defined by the `ComponentDescription` data type. For more information on the fields of this record, see “The Component Description Record” beginning on page 6-37.

```
TYPE
    ComponentDescription =
      RECORD
         componentType:          OSType;      {type}
         componentSubType:       OSType;      {subtype}
         componentManufacturer:  OSType;      {manufacturer}
         componentFlags:         LongInt;     {control flags}
         componentFlagsMask:     LongInt;     {mask for flags}
      END;
```

By default, the Component Manager considers all fields of the component description record when performing a search. Your application can override the default behavior of which fields the Component Manager considers for a search. Specify 0 in any field of the component description record to prevent the Component Manager from considering the information in that field when performing the search.

Listing 6-1 shows an application-defined procedure, MyFindVideoComponent, that fills out a component description record to specify the search criteria for the desired component. The MyFindVideoComponent procedure then uses the FindNextComponent function to return the first component with the specified characteristics—in this example, any component with the type VideoDigitizerComponentType.

Listing 6-1 Finding a component

```
PROCEDURE MyFindVideoComponent(VAR videoCompID: Component);
VAR
   videoDesc: ComponentDescription;
BEGIN
   {find a video digitizer component}
   videoDesc.componentType := VideoDigitizerComponentType;
   videoDesc.componentSubType := OSType(0);     {any subtype}
   videoDesc.componentManufacturer:= OSType(0);{any manufacturer}
   videoDesc.componentFlags := 0;
   videoDesc.componentFlagsMask := 0;
   videoCompID := FindNextComponent(Component(0), videoDesc);
END;
```

The FindNextComponent function requires two parameters: a value that indicates which component to begin the search with and a component description record. You can specify 0 in the first parameter to start the search at the beginning of the component list. Alternatively, you can specify a component identifier obtained from a previous call to FindNextComponent.

The FindNextComponent function returns a component identifier to your application. The returned component identifier identifies a given component to the Component Manager. You can use this identifier to retrieve more information about the component or to open a connection to the component. The next two sections describe these tasks.

Opening a Connection to a Specific Component

You can open a connection to a specific component by calling the OpenComponent function (alternatively, you can use the OpenDefaultComponent function, as discussed in "Opening a Connection to a Default Component" on page 6-7). Your application must provide a component identifier to the OpenComponent function. You get a component identifier from the FindNextComponent function, as described in the previous section.

The `OpenComponent` function returns a component instance that identifies your connection to the component. Listing 6-2 shows how to use the `OpenComponent` function to gain access to a specific component. The application-defined procedure `MyGetComponent` uses the `MyFindVideoComponent` procedure (defined in Listing 6-1) to find a video digitizer component and then opens the component.

Listing 6-2 Opening a specific component

```
PROCEDURE MyGetComponent
                     (VAR videoCompInstance: ComponentInstance);
VAR
   videoCompID:          Component;
BEGIN
   {first find a video digitizer component}
   MyFindVideoComponent(videoCompID);
   {now open it}
   IF videoCompID <> NIL THEN
      videoCompInstance := OpenComponent(videoCompID);
END;
```

Getting Information About a Component

You can use the `GetComponentInfo` function to retrieve information about a component, including the component name, icon, and other information. Listing 6-3 shows an application-defined procedure that retrieves information about a video digitizer component.

Listing 6-3 Getting information about a component

```
PROCEDURE MyGetCompInfo (compName, compInfo, compIcon: Handle;
                         VAR videoDesc: ComponentDescription);
VAR
   videoCompID:   Component;
   myErr:         OSErr;
BEGIN
   {first find a video digitizer component}
   MyFindVideoComponent(videoCompID);
   {now get information about it}
   IF videoCompID <> NIL THEN
      myErr := GetComponentInfo(videoCompID, videoDesc, compName,
                                compInfo, compIcon);
END;
```

You specify the component in the first parameter to GetComponentInfo. You specify the component using either a component identifier (obtained from FindNextComponent or RegisterComponent) or a component instance (obtained from OpenDefaultComponent or OpenComponent).

The GetComponentInfo function returns information about the component in the second through fifth parameters of the function. The GetComponentInfo function returns information about the component (such as its type, subtype, and manufacturer) in a component description record. The function also returns the component name, icon, and other information through handles. You must allocate these handles before calling GetComponentInfo. (Alternatively, you can specify NIL in the compName, compInfo, and compIcon parameters if you do not want the information returned.) The icon returned in the compIcon parameter is a handle to a black-and-white icon. If a component has an icon family, you can retrieve a handle to its icon suite using GetComponentIconSuite.

Using a Component

Once you have established a connection to a component, you can use its services.

Each time you call a component routine, you must specify the component instance that identifies your connection and provide any other parameters as required by the routine.

For example, Listing 6-4 illustrates the use of a drawing component. The application-defined procedure establishes a connection to a drawing component, calls the component's DrawerSetup function to establish the rectangle in which to draw the desired object, and then draws the object using the DrawerDraw function.

Listing 6-4 Using a drawing component

```
PROCEDURE MyDrawAnOval (VAR aDrawOvalComp: ComponentInstance);
VAR
   r:              Rect;
   result:         ComponentResult;
BEGIN
   {open a connection to a drawing component}
   aDrawOvalComp := OpenDefaultComponent('draw', 'oval');
   IF aDrawOvalComp <> NIL THEN
   BEGIN
      SetRect(r, 40, 40, 80, 80);
      {set up rectangle for oval}
      result := DrawerSetup(aDrawOvalComp, r);
      IF result = noErr THEN
         result := DrawerDraw(aDrawOvalComp);{draw oval}
   END;
END;
```

If you specify an invalid connection as a parameter to a component routine, the Component Manager sets the function result of the component routine to `badComponentInstance`.

Each component type supports a defined set of functions. You must refer to the appropriate documentation for a description of the functions supported by a component. You also need to refer to the component's documentation for information on the appropriate interface files that you must include to use the component (the interface files for the drawing component are shown beginning on page 6-28). The components that Apple provides with QuickTime are described in *Inside Macintosh: QuickTime Components*. As an example, drawing components support the following functions:

```
FUNCTION DrawerSetup(myInstance: ComponentInstance;
                     VAR r: Rect): ComponentResult;
FUNCTION DrawerClick(myInstance: ComponentInstance;
                     p: Point): ComponentResult;
FUNCTION DrawerMove (myInstance: ComponentInstance; x: Integer;
                     y: Integer): ComponentResult;
FUNCTION DrawerDraw (myInstance: ComponentInstance)
                     : ComponentResult;
FUNCTION DrawerErase(myInstance: ComponentInstance)
                     : ComponentResult;
```

Closing a Connection to a Component

When you finish using a component, you must close your connection to that component. Use the `CloseComponent` function to close the connection. For example, this code calls the application-defined procedure `MyDrawAnOval` (see Listing 6-4), which opens a connection to a drawing component and uses that component to draw an oval. This code closes the oval drawer component after it is finished using it.

```
VAR
   aDrawOvalComp: ComponentInstance;
   result:        OSErr;

MyDrawAnOval(aDrawOvalComp);      {open component and draw an oval}
result := DrawerErase(aDrawOvalComp);  {erase the oval}
result := CloseComponent(aDrawOvalComp); {close the component}
```

Creating Components

This section describes how to create a component and how your component interacts with the Component Manager. This section also describes many of the routines that the Component Manager provides to help you manage your component. If you are developing a component, you should read the material in this section.

If you are developing an application that uses components, you may find this material interesting, but you do not need to be familiar with it. You should read the preceding section, "Using the Component Manager," and then use the "Component Manager Reference" section as needed.

This section discusses how you can

- structure your component
- respond to requests from the Component Manager
- define the functions that applications may call to request service from your component
- manage your component with the help of the Component Manager
- make your component available for use by applications

This section presents several examples demonstrating how to create components and register them with the Component Manager. All of these examples are based on a "drawing component"—a simple component that draws an object of a particular shape on the screen. This section includes the code for a drawing component.

The Structure of a Component

Every component must have a single entry point that returns a value of type `ComponentResult` (a long integer). Whenever the Component Manager receives a request for your component, it calls your component's entry point and passes any parameters, along with information about the current connection, in a component parameters record. The Component Manager also passes a handle to the global storage (if any) associated with that instance of your component.

When your component receives a request, it should examine the parameters to determine the nature of the request, perform the appropriate processing, set an error code if necessary, and return an appropriate function result to the Component Manager.

The component parameters record is defined by a data structure of type `ComponentParameters`.

```
TYPE ComponentParameters =
      PACKED RECORD
         flags:      Char;                   {reserved}
         paramSize:  Char;                   {size of parameters}
         what:       Integer;                {request code}
         params:     ARRAY[0..0] OF LongInt; {actual parameters}
      END;
```

The `what` field contains a value that specifies the type of request. Negative values are reserved for definition by Apple. You can use values greater than or equal to 0 to define other requests that are supported by your component. Follow these guidelines when defining your request codes: request codes between 0 and 256 are reserved for definition by components of a given type and a given type-subtype. Use request codes greater than 256 for requests that are unique to your component. For example, a certain component of a certain type-subtype might support values 0 through 5 as requests that are supported by all components of that type, values 128 through 140 as requests that are supported by all components of that given type-subtype, and values 257 through 260 as requests supported only by that component.

Table 6-1 shows the request codes defined by Apple and the actions your component should take upon receiving them. Note that four of the request codes—open, close, can do, and version—are required. Your component must respond to these four request codes. These request codes are described in greater detail in "Handling Requests for Service" beginning on page 6-18.

Table 6-1 Request codes

Request code	Action your component should perform	Required
`kComponentOpenSelect`	Open a connection	Yes
`kComponentCloseSelect`	Close an open connection	Yes
`kComponentCanDoSelect`	Determine whether your component supports a particular request	Yes
`kComponentVersionSelect`	Return your component's version number	Yes
`kComponentRegisterSelect`	Determine whether your component can operate in the current environment	No

Table 6-1 Request codes (continued)

Request code	Action your component should perform	Required
`kComponentTargetSelect`	Call another component whenever it would call itself (as a result of your component being used by another component)	No
`kComponentUnregisterSelect`	Perform any operations that are necessary as a result of your component being unregistered	No

The example drawing component (shown in Listing 6-5 on page 6-16) supports the four required request codes, and in addition supports the request codes that are required for all components of the type `'draw'`. All drawing components must support these request codes:

```
CONST
      kDrawerSetUpSelect      = 0;  {set up drawing region}
      kDrawerDrawSelect       = 1;  {draw the object}
      kDrawerEraseSelect      = 2;  {erase the object}
      kDrawerClickSelect      = 3;  {determine if cursor is }
                                    { inside of the object}
      kDrawerMoveSelect       = 4;  {move the object}
```

The `params` field of the component parameters record is an array that contains the parameters specified by the application that called your component. You can directly extract the parameters from this array, or you can use the `CallComponentFunction` or `CallComponentFunctionWithStorage` function to extract the parameters from this array and pass these parameters to a subroutine of your component (see page 6-63 and page 6-64 for more information about these functions).

Listing 6-5 shows the structure of a drawing component—a simple component that draws an object on the screen. The component subtype of a drawing component indicates the type of object the component draws. This particular drawing component is of the subtype `'oval'`; it draws oval objects.

Whenever an application calls your component, the Component Manager calls your component's main entry point (for example, the `OvalDrawer` function). This entry point must be the first function in the component's code segment.

As previously described, the Component Manager passes two parameters to your component: a component parameters record and a parameter of type `Handle`.
The parameters specified by the calling application are contained in the component parameters record. Your component can access the parameters directly from this record. Alternatively, as shown in Listing 6-5, you can use Component Manager routines to extract the parameters from this array and invoke a subroutine of your component. By taking advantage of these routines, you can simplify the structure of your component code.

The OvalDrawer function first examines the value of the what field of the component parameters record. The what field contains the request code. The OvalDrawer function performs the action specified by the request code. The OvalDrawer function uses a number of subroutines to carry out the desired action. It uses the Component Manager routines CallComponentFunction and CallComponentFunctionWithStorage to extract the parameters from the component parameters record and to call the specified component's subroutine with these parameters.

For example, when the drawing component receives the request code kComponentOpenSelect, it calls the function CallComponentFunction. It passes the component parameters record and a pointer to the component's OvalOpen subroutine as parameters to CallComponentFunction. This function extracts the parameters and calls the OvalOpen function. The OvalOpen function allocates memory for this instance of the component. Your component can allocate memory to hold global data when it receives an open request. To do this, allocate the memory and then call the SetComponentInstanceStorage function. This function associates the allocated memory with the current instance of your component. The next time this instance of your component is called, the Component Manager passes a handle to your previously allocated memory in the storage parameter. For additional information on handling the open request, see "Responding to the Open Request" on page 6-19.

When the drawing component receives the drawing setup request (indicated by the kDrawerSetupSelect constant), it calls the Component Manager function CallComponentFunctionWithStorage. Like CallComponentFunction, this function extracts the parameters and calls the specified subroutine (OvalSetup) . The CallComponentFunctionWithStorage function also passes as a parameter to the subroutine a handle to the memory associated with this instance of the component. The OvalSetup subroutine can use this memory as needed. For additional information on handling the drawing setup request, see "Responding to Component-Specific Requests" on page 6-26.

Listing 6-5 A drawing component for ovals

```
UNIT Ovals;
INTERFACE
{include a USES statement if required}

FUNCTION OvalDrawer (params: ComponentParameters;
                     storage: Handle): ComponentResult;

IMPLEMENTATION

CONST
   kOvalDrawerVersion      = 0;  {version number of this component}

   kDrawerSetUpSelect      = 0;  {set up drawing region}
```

```
   kDrawerDrawSelect       = 1;  {draw the object}
   kDrawerEraseSelect      = 2;  {erase the object}
   kDrawerClickSelect      = 3;  {determine if cursor is }
                                 { inside of the object}
   kDrawerMoveSelect       = 4;  {move the object}
TYPE
   GlobalsRecord =
      RECORD
         bounds:              Rect;
         boundsRgn:           RgnHandle;
         self:                ComponentInstance;
      END;
   GlobalsPtr      = ^GlobalsRecord;
   GlobalsHandle   = ^GlobalsPtr;

{any subroutines used by the component go here}

FUNCTION OvalDrawer (params: ComponentParameters;
                     storage: Handle): ComponentResult;
BEGIN
   {perform action corresponding to request code}
   IF params.what < 0 THEN    {handle the required request codes}
      CASE (params.what) OF
      kComponentOpenSelect:
         OvalDrawer := CallComponentFunction(params,
                                          ComponentRoutine(@OvalOpen));
      kComponentCloseSelect:
         OvalDrawer := CallComponentFunctionWithStorage(storage, params,
                                           ComponentRoutine(@OvalClose));
      kComponentCanDoSelect:
         OvalDrawer := CallComponentFunction(params,
                                          ComponentRoutine(@OvalCanDo));
      kComponentVersionSelect:
         OvalDrawer := kOvalDrawerVersion;
      OTHERWISE
         OvalDrawer := badComponentSelector;
      END {of CASE}
   ELSE                    {handle component-specific request codes}
      CASE (params.what) OF
      kDrawerSetupSelect:
         OvalDrawer := CallComponentFunctionWithStorage
                                          (storage, params,
                                           ComponentRoutine(@OvalSetup));
```

```
      kDrawerDrawSelect:
         OvalDrawer := CallComponentFunctionWithStorage
                                                     (storage, params,
                                                      ComponentRoutine(@OvalDraw));
      kDrawerEraseSelect:
         OvalDrawer := CallComponentFunctionWithStorage
                                                     (storage, params,
                                                      ComponentRoutine(@OvalErase));
      kDrawerClickSelect:
         OvalDrawer := CallComponentFunctionWithStorage
                                                     (storage, params,
                                                      ComponentRoutine(@OvalClick));
      kDrawerMoveSelect:
         OvalDrawer := CallComponentFunctionWithStorage
                                                     (storage, params,
                                                      ComponentRoutine(@OvalMoveTo));
     OTHERWISE
         OvalDrawer := badComponentSelector;
   END;  {of CASE}
END; {of OvalDrawer}

END.
```

The next section describes how your component should respond to the required request codes. Following sections provide more information on

- defining your component's interfaces
- registering your component
- how to store your component in a component resource file

Handling Requests for Service

Whenever an application requests services from your component, the Component Manager calls your component and passes two parameters: the application's parameters in a component parameters record and a handle to the memory associated with the current connection. The component parameters record also contains information identifying the nature of the request.

There are two classes of requests: requests that are defined by the Component Manager and requests that are defined by your component. The Component Manager defines seven request codes: open, close, can do, version, register, unregister, and target. All components must support open, close, can do, and version requests. The register, unregister, and target requests are optional. Apple reserves all negative request codes for definition by the Component Manager. You are free to assign request codes greater than or equal to 0 to the functions supported by a component whose interface you have

defined. (However, request codes between 0 and 256 are reserved for definition by components of a given type and a given type-subtype. Request codes greater than 256 are available for requests that are unique to your component.)

You can refer to the standard request codes with these constants.

```
CONST kComponentOpenSelect       = -1; {open request}
      kComponentCloseSelect      = -2; {close request}
      kComponentCanDoSelect      = -3; {can do request}
      kComponentVersionSelect    = -4; {version request}
      kComponentRegisterSelect   = -5; {register request}
      kComponentTargetSelect     = -6; {target request}
      kComponentUnregisterSelect = -7; {unregister request}
```

The following sections discuss what your component must do when it receives these Component Manager requests.

Responding to the Open Request

The Component Manager issues an open request to your component whenever an application or any other client tries to open a connection to your component by calling the `OpenComponent` (or `OpenDefaultComponent`) function. The open request allows your component to establish the environment to support the new connection. Your component must support this request.

Your component should perform the necessary processing to establish the new connection. At a minimum, you must allocate the memory for any global data for the connection. Be sure to allocate this memory in the current heap zone, not in the system heap. You should call the `SetComponentInstanceStorage` procedure to inform the Component Manager that you have allocated memory. The Component Manager stores a handle to the memory and provides that handle to your component as a parameter in subsequent requests.

You may also want to open and read data from your component's resource file—if you do so, use the `OpenComponentResFile` function to open the file and be sure to close the resource file before returning.

If your component uses the services of other components, open connections to them when you receive the open request.

Once you have successfully set up the connection, set your component's function result to 0 and return to the Component Manager.

You can also refuse the connection. If you cannot successfully establish the environment for a connection (for example, there is insufficient memory to support the connection, or required hardware is unavailable), you can refuse the connection by setting the component's function result to a nonzero value. You can also use the open request as an opportunity to restrict the number of connections your component can support.

If your application is registered globally, you should also set the A5 world for your component in response to the open request. You can do this using the

SetComponentInstanceA5 procedure. (See page 6-68 for information on this procedure.)

The Component Manager sets these fields in the component parameters record that it provides to your component on an open request:

Field descriptions

what	The Component Manager sets this field to kComponentOpenSelect.
params	The first entry in this array contains the component instance that identifies the new connection.

Listing 6-6 shows the subroutine that handles the open request for the drawing component. Note that your component can directly access the parameters from the component parameters record, or use subroutines and the CallComponentFunction and CallComponentFunctionWithStorage functions to extract the parameters for you (see Listing 6-5 on page 6-16). The code in this chapter takes the second approach.

The OvalOpen function allocates memory to hold global data for this instance of the component. It calls the SetComponentInstanceStorage function so that the Component Manager can associate the allocated memory with this instance of the component. The Component Manager passes a handle to this memory in subsequent calls to this instance of the component.

Listing 6-6 Responding to an open request

```
FUNCTION OvalOpen (self: ComponentInstance): ComponentResult;
VAR
   myGlobals: GlobalsHandle;
BEGIN
   {allocate storage}
   myGlobals :=
            GlobalsHandle(NewHandleClear(sizeof(GlobalsRecord)));
   IF myGlobals = NIL THEN
      OvalOpen := MemError
   ELSE
   BEGIN
      myGlobals^^.self := self;
      myGlobals^^.boundsRgn := NewRgn;
      SetComponentInstanceStorage(myGlobals^^.self,
                                  Handle(myGlobals));
      {if your component is registered globally, set }
      { its A5 world before returning}
      OvalOpen := noErr;
   END;
END;
```

Responding to the Close Request

The Component Manager issues a close request to your component when a client application closes its connection to your component by calling the `CloseComponent` function. Your component should dispose of the memory associated with the connection. Your component must support this request. Your component should also close any files or connections to other components that it no longer needs.

The Component Manager sets these fields in the component parameters record that it provides to your component on a close request:

Field descriptions

`what`	The Component Manager sets this field to `kComponentCloseSelect`.
`params`	The first entry in this array contains the component instance that identifies the open connection.

Listing 6-7 shows the subroutine that handles the close request for the drawing component (as defined in Listing 6-5 on page 6-16). The `OvalClose` function closes the open connection. The drawing component uses the `CallComponentFunctionWithStorage` function to call the `OvalClose` function (see Listing 6-5). Because of this, in addition to the parameters specified in the component parameters record, the Component Manager also passes to the `OvalClose` function a handle to the memory associated with the component instance.

Listing 6-7 Responding to a close request

```
FUNCTION OvalClose (globals: GlobalsHandle;
                    self: ComponentInstance): ComponentResult;
BEGIN
   IF globals <> NIL THEN
   BEGIN
      DisposeRgn(globals^^.boundsRgn);
      DisposeHandle(Handle(globals));
   END;
   OvalClose := noErr;
END;
```

IMPORTANT

When responding to a close request, you should always test the handle passed to your component against `NIL` because it is possible for your close request to be called with a `NIL` handle in the `storage` parameter. For example, you can receive a `NIL` handle if your component returns a nonzero function result in response to an open request. ▲

Responding to the Can Do Request

The Component Manager issues a can do request to your component when an application calls the `ComponentFunctionImplemented` function to determine whether your component supports a given request code. Your component must support this request.

Set your component's function result to 1 if you support the request code; otherwise, set your function result to 0.

The Component Manager sets these fields in the component parameters record that it provides to your component on a can do request:

Field descriptions

`what` The Component Manager sets this field to `kComponentCanDoSelect`.

`params` The first entry in this array contains the request code as an integer value.

Listing 6-8 shows the subroutine that handles the can do request for the drawing component (as defined in Listing 6-5 on page 6-16). The `OvalCanDo` function examines the specified request code and compares it with the request codes that it supports. It returns a function result of 1 if it supports the request code; otherwise, it returns 0.

Listing 6-8 Responding to the can do request

```
FUNCTION OvalCanDo (selector: Integer): ComponentResult;
BEGIN
   IF ((selector >= kComponentVersionSelect) AND
      (selector <= kDrawerMoveSelect)) THEN
      OvalCanDo := 1                {valid request}
   ELSE
      OvalCanDo := 0;               {invalid request}
END;
```

Responding to the Version Request

The Component Manager issues a version request to your component when an application calls the `GetComponentVersion` function to retrieve your component's version number. Your component must support this request.

In response to a version request, your component should return its version number as its function result. Use the high-order 16 bits to represent the major version and the low-order 16 bits to represent the minor version. The major version should represent the component specification level; the minor version should represent your implementation's version number.

If the Component Manager supports automatic version control (a feature available in version 3 and above of the manager), it automatically resolves conflicts between different versions of the same component. For more information on this feature, see the next section, "Responding to the Register Request."

The Component Manager sets only the `what` field in the component parameters record that it provides to your component on a version request:

Field description

`what` The Component Manager sets this field to `kComponentVersionSelect`.

Listing 6-5 on page 6-16 shows how the drawing component handles the version request. It simply returns its version number as its function result.

Responding to the Register Request

The Component Manager may issue a register request when your component is registered. This request gives your component an opportunity to determine whether it can operate in the current environment. For example, your component might use the register request to verify that a specific piece of hardware is available on the computer. This is an optional request—your component is not required to support it.

The Component Manager issues this request only if you have set the `cmpWantsRegisterMessage` flag to 1 in the `componentFlags` field of your component's component description record (see "Data Structures for Components" beginning on page 6-52 for more information about the component description record).

Your component should not normally allocate memory in response to the register request. The register request is provided so that your application can determine whether it should be registered and made available to any clients. Once a client attempts to connect to your component, your component receives an open request, at which time it can allocate any required memory. Because your component might not be opened during a particular session, following this guideline allows other applications to make use of memory that isn't currently needed by your component.

If you want the Component Manager to provide automatic version control (a feature available in version 3 and above of the manager), your component can specify the `componentDoAutoVersion` flag in the optional extension to the component resource. If you specify this flag, the Component Manager registers your component only if there is no later version available. If an older version is already registered, the Component Manager unregisters it. If a newer version of the same component is registered after yours, the Component Manager automatically unregisters your component. You can use this automatic version control feature to make sure that the most recent version of your component is registered, regardless of the number of versions that are installed.

Set your function result to `TRUE` to indicate that you do not want your component to be registered; otherwise, set the function result to `FALSE`.

The Component Manager sets only the `what` field in the component parameters record that it provides to your component on a register request:

Field description

`what` The Component Manager sets this field to `kComponentRegisterSelect`.

If you request that your component receive a register request, the Component Manager actually sends your component a series of three requests: an open request, then the register request, followed by a close request.

For more information about the process the Component Manager uses to register components, see "Registering a Component" on page 6-30.

Responding to the Unregister Request

The unregister request is supported only in version 3 and above of the Component Manager. If your component specifies the `componentWantsUnregister` flag in the `componentRegisterFlags` field of the optional extension to the component resource, the Component Manager may issue an unregister request when your component is unregistered. This request gives your component an opportunity to perform any clean-up operations, such as resetting the hardware. This is an optional request—your component is not required to support it.

Return any error information as your component's function result.

The Component Manager sets only the `what` field in the component parameters record that it provides to your component on an unregister request:

Field description

`what` The Component Manager sets this field to `kComponentUnregisterSelect`.

If you have specified that your component should not receive a register request, then your component does not receive an unregister request if it has not been opened. However, if a client opens and closes your component, and then later the Component Manager unregisters your component, the Component Manager does send your component an unregister request (in a series of three requests: open, unregister, close).

If you have specified that your component should receive a register request, when your component is registered the Component Manager sends your component a series of three requests: an open request, then the register request, followed by a close request. In this situation, even if your component is not opened by a client, the Component Manager sends your component an unregister request when it unregisters your component.

For more information about the componentWantsUnregister flag, see “Resources” beginning on page 6-80.

Responding to the Target Request

The Component Manager issues a target request to inform an instance of your component that it has been targeted by another component. The component that targets another component instance may also choose to first capture the component, but it is not necessary to do so. Thus, a component can choose to

- capture a component and target an instance of it
- capture a component without targeting any instance of it
- target a component instance without capturing the component

To first capture another component, the capturing component calls the CaptureComponent function. When a component is captured, the Component Manager removes it from the list of available components. This makes the captured component available only to the capturing component and to any clients currently connected to it. Typically, a component captures another component when it wants to override one or more functions of the other component.

After calling the CaptureComponent function, the capturing component can choose to target a particular instance of the component. However, a component can capture another component without targeting it.

A component uses the ComponentSetTarget function to send a target request to a specific component instance. After receiving a target request, whenever the targeted component instance would call itself (that is, call any of its defined functions), instead it should always call the component that targeted it.

For example, a component called NewMath might first capture a component called OldMath. NewMath does this by using FindNextComponent to get a component identifier for OldMath. NewMath then calls CaptureComponent to remove OldMath from the list of available components. At this point, no other clients can access OldMath, except for those clients previously connected to it.

NewMath might then call ComponentSetTarget to target a particular component instance of OldMath. The ComponentSetTarget function sends a target request to the specified component instance. When OldMath receives a target request, it saves the component instance of the component that targeted it. When OldMath receives a request, it processes it as usual. However, whenever OldMath calls one of its defined functions: in its defined API, it calls NewMath instead. (Suppose OldMath provides request codes for these functions: DoMultiply, DoAdd, DoDivide, and DoSubtract. If OldMath's DoMultiply function calls its own DoAdd function, then OldMath calls NewMath to perform the addition.)

The target request is an optional request—your component is not required to support it.

The Component Manager sets these fields in the component parameters record that it provides to your component on a target request:

Field descriptions

`what` — The Component Manager sets this field to `kComponentTargetSelect`.

`params` — The first entry in this array contains the component instance that identifies the component issuing the target request.

Responding to Component-Specific Requests

When your component receives a component-specific request, it should handle the request as appropriate. For example, the drawing component responds to five component-specific requests: setup, draw, erase, click, and move to. See Listing 6-5 on page 6-16 for the code that defines the drawing component's entry point. The drawing component uses `CallComponentFunctionWithStorage` to extract the parameters and call the appropriate subroutine.

Listing 6-9 shows the drawing component's `OvalSetup` function. This function sets up the data structures that must be in place before drawing the oval.

Listing 6-9 Responding to the setup request

```
FUNCTION OvalSetup (globals: GlobalsHandle;
                    boundsRect: Rect): ComponentResult;
VAR
   ignoreErr: ComponentResult;
BEGIN
   globals^^.bounds := boundsRect;
   OpenRgn;
   ignoreErr := OvalDraw(globals);
   CloseRgn(globals^^.boundsRgn);
   OvalSetup := noErr;
END;
```

Listing 6-10 shows the drawing component's `OvalDraw` function. This function draws an oval in the previously allocated region.

Listing 6-10 Responding to the draw request

```
FUNCTION OvalDraw (globals: GlobalsHandle): ComponentResult;
BEGIN
   FrameOval(globals^^.bounds);
   OvalDraw := noErr;
END;
```

Listing 6-11 shows the drawing component's `OvalErase` function. This function erases an oval.

Listing 6-11 Responding to the erase request

```
FUNCTION OvalErase (globals: GlobalsHandle): ComponentResult;
BEGIN
   EraseOval(globals^^.bounds);
   OvalErase := noErr;
END;
```

Listing 6-12 shows the drawing component's `OvalClick` function. This function determines whether the given point is within the oval. If so, the function returns 1; otherwise, it returns 0. Because the `OvalClick` function returns information other than error information as its function result, `OvalClick` sets any error information using `SetComponentInstanceError`.

Listing 6-12 Responding to the click request

```
FUNCTION OvalClick (globals: GlobalsHandle; p: Point)
                    : ComponentResult;
BEGIN
   IF PtInRgn(p, globals^^.boundsRgn) THEN
      OvalClick := 1
   ELSE
      OvalClick := 0;
   SetComponentInstanceError(globals^^.self, noErr);
END;
```

Listing 6-13 shows the drawing component's `OvalMoveTo` function. This function moves the oval's coordinates to the specified location. Note that this function does not erase or draw the oval; the calling application is responsible for issuing the appropriate requests. For example, the calling application can issue requests to draw, erase, move to, and draw—to draw the oval in one location, then erase the oval, move it to a new location, and finally draw the oval in its new location.

Listing 6-13 Responding to the move to request

```
FUNCTION OvalMoveTo (globals: GlobalsHandle; x, y: Integer)
                     : ComponentResult;
VAR
   r: Rect;
BEGIN
   r := globals^^.bounds;
   x := x - (r.right + r.left) DIV 2;
   y := y - (r.bottom + r.top) DIV 2;
   OffsetRect(globals^^.bounds, x, y);
   OffsetRgn(globals^^.boundsRgn, x, y);
   OvalMoveTo := noErr;
END;
```

Reporting an Error Code

The Component Manager maintains error state information for all currently active connections. In general, your component returns error information in its function result; a nonzero function result indicates an error occurred, and a function result of 0 indicates the request was successful. However, some requests require that your component return other information as its function result. In these cases, your component can use the `SetComponentInstanceError` procedure to report its latest error state to the Component Manager. You can also use this procedure at any time during your component's execution to report an error.

Defining a Component's Interfaces

You define the interfaces supported by your component by declaring a set of functions for use by applications. These function declarations specify the parameters that must be provided for each request. The following code illustrates the general form of these function declarations, using the setup request defined for the sample drawing component as an example:

```
FUNCTION DrawerSetup (myInstance: ComponentInstance;
                      VAR r: Rect): ComponentResult;
```

This example declares a function that supports the setup request. The first parameter to any component function must be a parameter that accepts a component instance. The Component Manager uses this value to correctly route the request. The calling application must supply a valid component instance when it calls your component. The second and following parameters are those required by your component function. For example, the `DrawerSetup` function takes one additional parameter, a rectangle. Finally, all component functions must return a function result of type `ComponentResult` (a long integer).

These function declarations must also include inline code. This code identifies the request code assigned to the function, specifies the number of bytes of parameter data accepted by the function, and executes a trap to the Component Manager. To continue with the Pascal example used earlier, the inline code for the `DrawerSetup` function is

```
INLINE $2F3C, $0004, $0000, $7000, $A82A;
```

The first element of this code, `$2F3C`, is the opcode for a move instruction that loads the contents of the next two elements onto the stack. The Component Manager uses these values when it invokes your component.

The second element, `$0004`, defines an integer value that specifies the number of bytes of parameter data required by the function, not including the component instance parameter. In this case, the size of a pointer to the rectangle is specified: 4 bytes.

Note

Note that Pascal calling conventions require that Boolean and 1-byte parameters are passed as 16-bit integer values. ◆

The third element, `$0000`, specifies the request code for this function as an integer value. Each function supported by your component must have a unique request code. Your component uses this request code to identify the application's request. You may define only request code values greater than or equal to 0; negative values are reserved for definition by the Component Manager. Recall from the oval drawing component that the request code for the setup request, `kDrawerSetUpSelect`, has a value of 0.

The fourth element, `$7000`, is the opcode for an instruction that sets the D0 register to 0, which tells the Component Manager to call your component rather than to field the request itself.

The fifth element, `$A82A`, is the opcode for an instruction that executes a trap to the Component Manager.

If you are declaring functions for use by Pascal-language applications, your declarations should take the following form:

```
FUNCTION DrawerSetup (myInstance: ComponentInstance;
                      VAR r: Rect): ComponentResult;
INLINE $2F3C, $0004, $0000, $7000, $A82A;
```

If you are declaring functions for use by C-language applications, your declarations can take the following form:

```
pascal ComponentResult DrawerSetup
                     (ComponentInstance myInstance, Rect *r) =
                           {0x2F3C,0x4,0x0,0x7000,0xA82A};
```

Alternatively, you can define the following statement to replace the inline code:

```
#define ComponentCall (callNum, paramSize)
                      {0x2F3C,paramSize,callNum,0x7000,0xA82A}
```

Using this statement results in the following declaration format:

```
pascal ComponentResult DrawerSetup
                  (ComponentInstance myInstance, Rect *r) =
                   ComponentCall (kDrawerSetUpSelect, 4);
```

When a client application calls your function, the system executes the inline code, which invokes the Component Manager. The Component Manager then formats a component parameters record, locates the storage for the current connection, and invokes your component. The Component Manager provides the component parameters record and a handle to the storage of the current connection to your component as function parameters.

Managing Components

This section discusses the Component Manager routines that help you manage your component. It describes how to register your component and how to allow applications to connect to your component.

Registering a Component

Applications must use the services of the Component Manager to find components that meet their needs. Before an application can find a component, however, that component must be registered with the Component Manager. When you register your component, the Component Manager adds the component to its list of available components.

There are two mechanisms for registering a component with the Component Manager. First, during startup processing, the Component Manager searches the Extensions folder (and all of the folders within the Extensions folder) for files of type 'thng'. If the file contains all the information needed for registration (see "Creating a Component Resource" on page 6-32 for more information on creating a component file), the Component Manager automatically registers the component stored in the file. Components registered in this manner are registered globally; that is, the component is made available to all applications and other clients.

Second, your application (or another application) can register your component. When you register your component in this manner, you can specify whether the component should be made available to all applications (global registration) or only to your application (local registration). Your application can register a component that is in memory or that is stored in a resource. You use the `RegisterComponent` function to register a component that is in memory. You use the `RegisterComponentResource` function to register a component that is stored in a component resource. See "The Component Resource" on page 6-80 for a description of the format and content of component resources. The code in Listing 6-14 demonstrates how an application can use the `RegisterComponent` function to register a component that is in memory.

Listing 6-14 Registering a component

```
VAR
   cd:       ComponentDescription;
   draw:     Component;

   WITH cd DO
   BEGIN          {initialize the component description record}
      componentType := 'draw';
      componentSubtype := 'oval';
      componentManufacturer := 'appl';
      componentFlags := 0;
      componentFlagsMask := 0;
   END;
   {register the component}
   draw := RegisterComponent(cd, ComponentRoutine(@OvalDrawer),
                             0, NIL, NIL, NIL);
```

The code in Listing 6-14 specifies six parameters to the `RegisterComponent` function. The first three are a component description record, a pointer to the component's entry point, and a value of 0 to indicate that this component should be made available only to this application. A component that is registered locally is visible only within the A5 world of the registering program. The last three parameters are specified as `NIL` to indicate that the component doesn't have a name, an information string, or an icon. See page 6-57 for more information on the `RegisterComponent` function.

When a component is registered and the `cmpWantsRegisterMessage` bit is not set in the `componentFlags` field of the component description record, the Component Manager adds the component to its list of registered components. Whenever a client requests access to or information about a component (for example, by using `OpenDefaultComponent`, `FindNextComponent`, or `GetComponentInfo`), the Component Manager searches its list of registered components.

If a component's cmpWantsRegisterMessage bit is set, the Component Manager does not automatically add your component to its list of registered components. Instead, it sends your component a series of three requests: open, register, and close. If your component returns a nonzero value as its function result in response to the register request, your component is not added to the Component Manager's list of registered components. Thus, clients are not able to connect to or get information about your component. You might choose to set the cmpWantsRegisterMessage bit if, for example, your application requires specific hardware.

Alternatively, you can let your component be automatically registered. Your component can then check for any specific hardware requirements upon receiving an open request. This lets clients attempt to connect to your component and also lets them get information about your component. However, in most cases, if your component requires specific hardware to operate, you should set the cmpWantsRegisterMessage bit and respond to the register request appropriately.

If your component controls a hardware resource, you should register your component once for each hardware resource that is available (rather than registering once and allowing multiple instances of your component). This allows clients to easily determine how many hardware resources are available by using the FindNextComponent function. If you register a component multiple times, be sure that you specify a unique name for each registration.

If the feature is available, you can request that the Component Manager provide automatic version control for your component (this feature is available only in version 3 and above of the manager). To request automatic version control, specify the componentDoAutoVersion flag in the optional extension to the component resource. If you specify this flag, the Component Manager registers your component only if there is no later version available. If an older version is already registered, the Component Manager unregisters it. If a newer version of the same component is registered after yours, the Component Manager automatically unregisters your component. You can use this automatic version control feature to make sure that the most recent version of your component is registered, regardless of the number of versions that are installed.

Creating a Component Resource

You can create a component resource (a resource of type 'thng') in a component file. A component file is a file whose resource fork contains a component resource and other required resources for the component. If you store your component in a component file, either you can allow applications to use the RegisterComponentResource function to register your component as needed, or you can automatically register your component at startup by storing your component file in the Extensions folder.

A component file consists of

- a component description record that specifies the characteristics of your component (its type, subtype, manufacturer, and control flags)
- the resource type and resource ID of your component's code resource
- the resource type and resource ID of your component's name string
- the resource type and resource ID of your component's information string
- the resource type and resource ID of your component's icon
- optional information about the component (its version number, additional flags, and resource ID of the component's icon family)
- the actual resources for your component's code, name, information string, and icon

Listing 6-15 shows, in Rez format, a component resource that defines an oval drawing component. This drawing component does not specify optional information (see Figure 6-5 on page 6-85 for the contents of the optional extension to the component resource). For compatibility with early versions of the Component Manager, component resources should be locked.

Listing 6-15 Rez input for a component resource

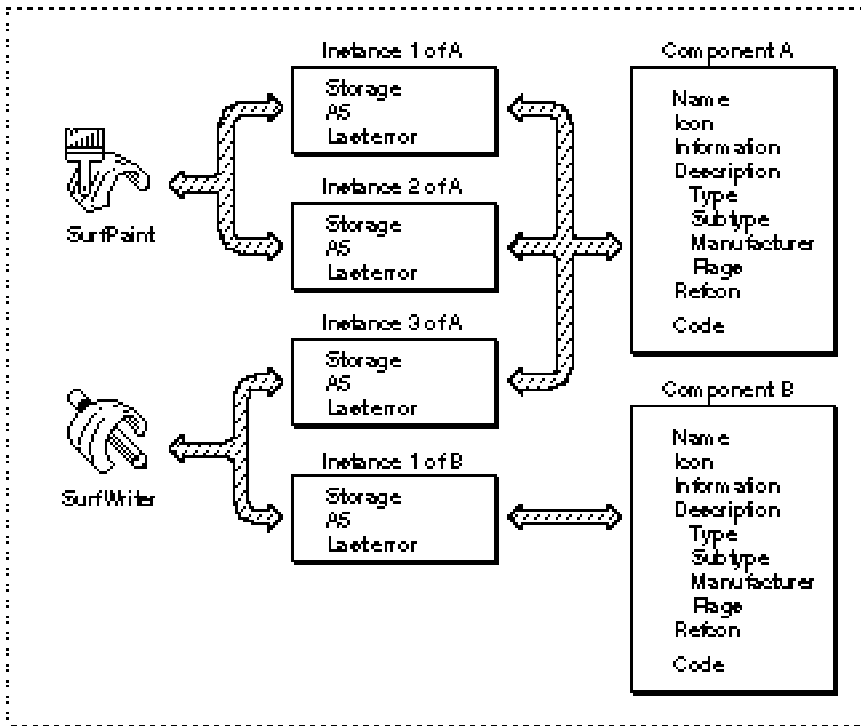
```
resource 'thng' (128, locked) {
   'draw',        /*component type*/
   'oval',        /*component subtype*/
   'appl',        /*component manufacturer*/
   $00000000,     /*component flags: 0*/
   $00000000,     /*reserved (component flags mask): 0*/
   'CODE',        /*component code resource type*/
   128,           /*component code resource ID*/
   'STR ',        /*component name resource type*/
   128,           /*component name resource ID*/
   'STR ',        /*component info resource type*/
   129,           /*component info resource ID*/
   'ICON',        /*component icon resource type*/
   128            /*component icon resource ID*/
   /*optional information (if any) goes here*/
};
```

The component resource, and the resources that define the component's code, name, information string, and icon, must be in the same file. A component file must have the file type `'thng'` and reside in the Extensions folder in order to be automatically registered by the Component Manager at startup.

Establishing and Managing Connections

Your component may support one or more connections at a time. In addition, a single application may have open connections with two or more different components at the same time. In fact, a single application can use more than one connection to a single component. Figure 6-2 shows two applications and two components: the first application, SurfPaint, uses two connections to component A; the second application, SurfWriter, uses one connection to component A and one to component B.

Figure 6-2 Supporting multiple component connections



A component can allocate separate storage for each open connection. A component can also set the A5 world for a specific component instance and can maintain separate error information for each instance. A component can also use a reference constant value to maintain global data for the component.

When an application requests that the Component Manager open a connection to your component, the Component Manager issues an open request to your component. At this time, your component should allocate any memory it needs in order to maintain a connection for the requesting application. Be sure to allocate this memory in the current heap zone rather than in the system heap. As described in "Responding to the Open Request" on page 6-19, you can use the `SetComponentInstanceStorage` procedure to associate the allocated memory with the component instance. Whenever the application requests services from your component, the Component Manager supplies

you with the handle to this memory. You can also use the open request as an opportunity to restrict the number of connections your component can support.

To allocate global data for your component, you can maintain a reference constant for use by your component. The Component Manager provides two routines, `SetComponentRefcon` and `GetComponentRefcon`, that allow you to work with your component's reference constant. Note that your component has one reference constant, regardless of the number of connections maintained by your component.

If your component uses its reference constant and is registered globally, be aware that in certain situations the Component Manager may clone your component. This situation occurs only when the Component Manager opens a component that is registered globally and there's no available space in the system heap. In this case, the Component Manager clones your component, creating a new registration of the component in the caller's heap, and returns to the caller the component identifier of the cloned component, not the component identifier of the original registration. The reference constant of the original component is not preserved in the cloned component. Thus you need to take extra steps to set the reference constant of the cloned component to the same value as that of the original component.

To determine whether your component has been cloned, you can examine your component's A5 world using the `GetComponentInstanceA5` function. If the returned value of the A5 world is nonzero, your component is cloned (only components registered globally can be cloned; if your component is registered locally it has a valid, nonzero A5 world and you don't need to check whether it's been cloned). If you determine that your component is cloned, you can retrieve the original reference constant by using the `FindNextComponent` function to iterate through all registrations of your component. You should compare the component identifier of the cloned component with the component identifier returned by `FindNextComponent`. Once you find a component with the same component description but a different component identifier, you've found the original component. You can then use `GetComponentRefcon` to get the reference constant of the original component and then use `SetComponentRefcon` to set the reference constant of the cloned component appropriately. This technique works if a component registers itself only once or registers itself multiple times but with a unique name for each registration. This technique does not work if a component registers itself multiple times using the same name.

When responding to a request from an application, your component can invoke the services of other components. The Component Manager provides two techniques for calling other components. First, your component can call another component using the standard mechanisms also used by applications. The Component Manager then passes the requests to the appropriate component, and your component receives the results of those requests.

Second, your component can redirect a request to another component. For example, you might want to create two similar components that provide different levels of service to applications. Rather than completely implementing both components, you could design one to rely on the capabilities of the other. Use the `DelegateComponentCall` function to pass a request on to another component.

Listing 6-16 shows an example of delegating a request to another component. The component in this example is a drawing component that draws rectangles. The `RectangleDrawer` component handles open, close, and setup requests. It delegates all other requests to another component. When the `RectangleDrawer` component receives an open request, it opens the component to which it will later delegate requests, and stores in its allocated storage the delegated component's component instance. It then specifies this value when it calls the `DelegateComponentCall` function.

Listing 6-16 Delegating a request to another component

```
FUNCTION RectangleDrawer(params: ComponentParameters;
                         storage: Handle): ComponentResult;
VAR
   theRtn:  ComponentRoutine;
   safe:    Boolean;
BEGIN
   safe := FALSE;
   CASE (params.what) OF
      kComponentOpenSelect:
         theRtn := ComponentRoutine(@RectangleOpen);
      kComponentCloseSelect:
         theRtn := ComponentRoutine(@RectangleClose);
      kDrawerSetupSelect:
         theRtn := ComponentRoutine(@RectangleSetup);
      OTHERWISE
      BEGIN
         safe := TRUE;
         IF (storage <> NIL) THEN
            RectangleDrawer :=
               DelegateComponentCall
                  (params,
                   ComponentInstance(StorageHdl(storage)^^.delegateInstance))
         ELSE
            RectangleDrawer := badComponentSelector;
      END;
   END; {of CASE}
   IF NOT safe THEN
      RectangleDrawer :=
         CallComponentFunctionWithStorage(storage, params, theRtn);
END;
```

Component Manager Reference

This section provides information about the data structures, routines, and resources defined by the Component Manager. This section is divided into the following topics:

- "Data Structures for Applications" describes the data structures used by applications.
- "Routines for Applications" discusses the Component Manager routines that are available to applications that use components.
- "Data Structures for Components" describes the data structures used by components.
- "Routines for Components" describes the Component Manager routines that are used by components.
- "Application-Defined Routine" describes how to define a component function and supply the appropriate registration information.
- "Resources" describes the format and content of component resources.

Assembly-Language Note

You can invoke Component Manager routines by using the trap `_ComponentDispatch` with the appropriate routine selector. The routine selectors are listed in "Assembly-Language Summary" beginning on page 6-98. ◆

Data Structures for Applications

This section describes the format and content of the data structures used by applications that use components.

Your application can use the component description record to find components that provide specific services or meet other selection criteria.

The Component Description Record

The component description record identifies the characteristics of a component, including the type of services offered by the component and its manufacturer.

Applications and components use component description records in different ways. An application that uses components specifies the selection criteria for finding a component in a component description record. A component uses the component description record to specify its registration information and capabilities. If you are developing a component, see page 6-52 for information on how a component uses the component description record.

The `ComponentDescription` data type defines the component description record.

```
TYPE  ComponentDescription =
      RECORD
         componentType:        OSType;      {type}
         componentSubType:     OSType;      {subtype}
         componentManufacturer:             {manufacturer}
                               OSType;
         componentFlags:       LongInt;     {control flags}
         componentFlagsMask:   LongInt;     {mask for control }
                                            { flags}
      END;
```

Field descriptions

`componentType`
: A four-character code that identifies the type of component. All components of a particular type must support a common set of interface routines. For example, drawing components all have a component type of `'draw'`.

  Your application can use this field to search for components of a given type. You specify the component type in the `componentType` field of the component description record you supply to the `FindNextComponent` or `CountComponents` routine. A value of 0 operates as a wildcard.

`componentSubType`
: A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component type. For example, the subtype of drawing components indicates the type of object the component draws. Drawing components that draw ovals have a subtype of `'oval'`.

  Your application can use the `componentSubType` field to perform a more specific lookup operation than is possible using only the `componentType` field. For example, you may want your application to use only components of a certain component type (`'draw'`) that also have a specific entry in the `componentSubType` field (`'oval'`). By specifying particular values for both fields in the component description record that you supply to the `FindNextComponent` or `CountComponents` routine, your application retrieves information about only those components that meet both of these search criteria. A value of 0 operates as a wildcard.

`componentManufacturer`
A four-character code that identifies the manufacturer of the component. This field allows for further differentiation between individual components. For example, components made by a specific manufacturer may support an extended feature set. Components provided by Apple use a manufacturer value of `'appl'`.

Your application can use this field to find components from a certain manufacturer. Specify the appropriate manufacturer code in the `componentManufacturer` field of the component description record you supply to the `FindNextComponent` or `CountComponents` routine. A value of 0 operates as a wildcard.

`componentFlags`
A 32-bit field that provides additional information about a particular component.

The high-order 8 bits are defined by the Component Manager. You should usually set these bits to 0.

The low-order 24 bits are specific to each component type. These flags can be used to indicate the presence of features or capabilities in a given component.

Your application can use these flags to further narrow the search criteria applied by the `FindNextComponent` or `CountComponents` routine. If you use the `componentFlags` field in a component search, you use the `componentFlagsMask` field to indicate which flags are to be considered in the search.

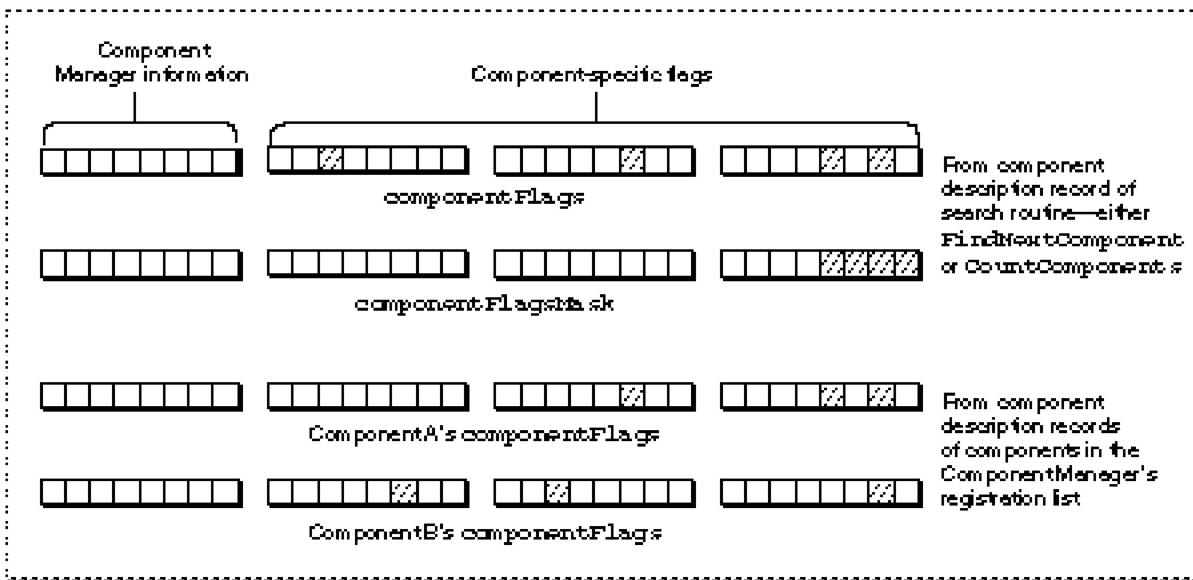
`componentFlagsMask`
A 32-bit field that indicates which flags in the `componentFlags` field are relevant to a particular component search operation.

For each flag in the `componentFlags` field that is to be considered as a search criterion by the `FindNextComponent` or `CountComponents` routine, your application should set the corresponding bit in the `componentFlagsMask` field to 1. The Component Manager considers only these flags during the search. You specify the desired flag value (either 0 or 1) in the `componentFlags` field.

For example, to look for a component with a specific control flag that is set to 0, set the appropriate bit in the `ComponentFlags` field to 0 and the same bit in the `ComponentFlagsMask` field to 1. To look for a component with a specific control flag that is set to 1, set the bit in the `ComponentFlags` field to 1 and the same bit in the `ComponentFlagsMask` field to 1. To ignore a flag, set the bit in the `ComponentFlagsMask` field to 0.

Figure 6-3 shows how the various fields interact during a search. In the case depicted in the figure, the `componentFlagsMask` field of a component description record supplied to a search routine specifies that only the low-order four flags of the `componentFlags` field are to be examined during the search. The `componentFlags` fields in the component description records of components A and B have a number of flags set. However, in this example the mask specifies that the Component Manager examine only the low-order 4 bits, and therefore only component A meets the search criteria.

Figure 6-3 Interaction between the `componentFlags` and `componentFlagsMask` fields



Component Identifiers and Component Instances

In general, when using Component Manager routines, your application must specify the particular component using either a component identifier or component instance. The Component Manager identifies *each component* by a component identifier. The Component Manager identifies *each instance* of a component by a component instance. Thus, when your application searches for a component with a particular type and subtype using the `FindNextComponent` function, `FindNextComponent` returns a component identifier that identifies the component. Similarly, your application specifies a component identifier to the `GetComponentInfo` function to obtain information about a component.

When you open a connection to a component, the `OpenDefaultComponent` and `OpenComponent` functions return a component instance. The returned component instance identifies that specific instance of the component. If you open the same component again, the Component Manager returns a different component instance. So a

component has a single component identifier and can have multiple component instances. To use a component function, your application specifies a component instance.

Although conceptually component identifiers and component instances serve different purposes, Component Manager routines (with the exception of `DelegateComponentCall`) allow you to use component identifiers and component instances interchangeably. If you do this, you must always coerce the data type appropriately.

A component identifier is defined by the data type `Component`:

```
TYPE
   {component identifier}
   Component         = ^ComponentRecord;
   ComponentRecord   =
   RECORD
      data: ARRAY[0..0] OF LongInt;
   END;
```

A component instance is defined by the data type `ComponentInstance`:

```
TYPE
   {component instance}
   ComponentInstance = ^ComponentInstanceRecord;
   ComponentInstanceRecord =
   RECORD
      data: ARRAY[0..0] OF LongInt;
   END;
```

Routines for Applications

This section discusses the Component Manager routines that are used by applications. If you are developing an application that uses components, you should read this section. If you are developing an application that registers components, you should also read “Registering Components” beginning on page 6-57.

If you are developing a component, you should read this section and “Routines for Components” beginning on page 6-56.

This section describes the routines that allow your application to

- search for components
- gain access to and release components
- get detailed information about specific components
- get component error information

Note
Any of the routines discussed in this section that require a component identifier also accept a component instance. Similarly, you can supply a component identifier to any routine that requires a component instance (except for the `DelegateComponentCall` function). If you do this, you must always coerce the data type appropriately. ◆

Finding Components

The Component Manager provides routines that allow your application to search for components. Your application specifies the search criteria in a component description record. (See "Data Structures for Applications" beginning on page 6-37 for information about the component description record.) Based on the values you specify in fields of the component description record, the Component Manager attempts to find components that meet the needs of your application.

You can use the `CountComponents` function to determine the number of components that match a component description. Use the `FindNextComponent` function to find an individual component that matches a description.

You can use the `GetComponentListModSeed` function to determine whether the list of registered components has changed.

FindNextComponent

The `FindNextComponent` function returns the component identifier for the next registered component that meets the selection criteria specified by your application. You specify the selection criteria in a component description record.

Your application can use the component identifier returned by this function to get more information about the component or to open the component.

```
FUNCTION FindNextComponent (aComponent: Component;
                            looking: ComponentDescription)
                            : Component;
```

`aComponent`
: The starting point for the search. Set this field to 0 to to start the search at the beginning of the component list. If you are continuing a search, you can specify a component identifier previously returned by the `FindNextComponent` function. The function then searches the remaining components.

`looking`
: A component description record. Your application specifies the criteria for the component search in the fields of this record.

: The Component Manager ignores fields in the component description record that are set to 0. For example, if you set all the fields to 0, all components meet the search criteria. In this case, your application can

retrieve information about all of the components that are registered in the system by repeatedly calling FindNextComponent and GetComponentInfo until the search is complete. Similarly, if you set all fields to 0 except for the componentManufacturer field, the Component Manager searches all registered components for a component supplied by the manufacturer you specify. Note that the FindNextComponent function does not modify the contents of the component description record you supply. To retrieve detailed information about a component, you need to use the GetComponentInfo function to get the component description record for each returned component.

DESCRIPTION

The FindNextComponent function returns the component identifier of a component that meets the search criteria. FindNextComponent returns a function result of 0 when there are no more matching components.

SEE ALSO

Use the GetComponentInfo function, described on page 6-48, to retrieve more information about a component. To open a component, use the OpenDefaultComponent or OpenComponent function, described on page 6-45 and page 6-46, respectively. See page 6-37 for information on the component description record.

See Listing 6-1 on page 6-9 for an example of searching for a specific component.

CountComponents

Your application can use the CountComponents function to determine the number of registered components that meet your selection criteria. You specify the selection criteria in a component description record. The CountComponents function returns the number of components that meet those search criteria.

```
FUNCTION CountComponents (looking: ComponentDescription): LongInt;
```

looking — A component description record. Your application specifies the criteria for the component search in the fields of this record.

The Component Manager ignores fields in the component description record that are set to 0. For example, if you set all the fields to 0, the Component Manager returns the number of components registered in the system. Similarly, if you set all fields to 0 except for the componentManufacturer field, the Component Manager returns the number of registered components supplied by the manufacturer you specify.

DESCRIPTION

The `CountComponents` function returns a long integer containing the number of components that meet the specified search criteria.

SEE ALSO

See page 6-37 for information on the component description record.

GetComponentListModSeed

The `GetComponentListModSeed` function allows you to determine if the list of registered components has changed. This function returns the value of the component registration seed number. By comparing this value to values previously returned by the this function, you can determine whether the list has changed. Your application may use this information to rebuild its internal component lists or to trigger other activity that is necessary whenever new components are available.

```
FUNCTION GetComponentListModSeed: LongInt;
```

DESCRIPTION

The `GetComponentListModSeed` function returns a long integer containing the component registration seed number. Each time the Component Manager registers or unregisters a component it generates a new, unique seed number.

Opening and Closing Components

The `OpenDefaultComponent`, `OpenComponent`, and `CloseComponent` functions allow your application to gain access to and release components. Your application must open a component before it can use the services provided by that component. Similarly, your application must close the component when it is finished using the component.

You can use the `OpenDefaultComponent` function to open a component of a specified component type and subtype. You do not have to supply a component description record or call the `FindNextComponent` function to use this function.

You use the `OpenComponent` function to gain access to a specified component. To use this function, your application must have previously obtained a component identifier for the desired component by using the `FindNextComponent` function. (If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.)

Once you are finished using a component, use the `CloseComponent` function to release the component.

OpenDefaultComponent

The `OpenDefaultComponent` function allows your application to gain access to the services provided by a component. Your application must open a component before it can call any component functions. You specify the component type and subtype values of the component to open. The Component Manager searches for a component that meets those criteria. If you want to exert more control over the selection process, you can use the `FindNextComponent` and `OpenComponent` functions.

```
FUNCTION OpenDefaultComponent (componentType: OSType;
                               componentSubType: OSType)
                               : ComponentInstance;
```

`componentType`
: A four-character code that identifies the type of component. All components of a particular type support a common set of interface routines. Your application uses this field to search for components of a given type.

`componentSubType`
: A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component type. For example, the subtype of an image compressor component indicates the compression algorithm employed by the compressor.

: Your application can use the `componentSubType` field to perform a more specific lookup operation than is possible using only the `componentType` field. For example, you may want your application to use only components of a certain component type (`'draw'`) that also have a specific subtype (`'oval'`). Set this parameter to 0 to select a component with any subtype value.

DESCRIPTION

The `OpenDefaultComponent` function searches its list of registered components for a component that meets the search criteria. If it finds a component that matches the search criteria, `OpenDefaultComponent` opens a connection to the component and returns a component instance. The returned component instance identifies your application's connection to the component. You must supply this component instance whenever you call the functions provided by the component. When you close the component, you must also supply this component instance to the `CloseComponent` function.

If more than one component in the list of registered components meets the search criteria, `OpenDefaultComponent` opens the first one that it finds in its list.

If it cannot open the specfied component, the `OpenDefaultComponent` function returns a function result of `NIL`.

SEE ALSO

For an example that opens a component using the `OpenDefaultComponent` function, see "Opening a Connection to a Default Component" beginning on page 6-7.

OpenComponent

The `OpenComponent` function allows your application to gain access to the services provided by a component. Your application must open a component before it can call any component functions. You specify the component with a component identifier that your application previously obtained from the `FindNextComponent` function.

Alternatively, you can use the `OpenDefaultComponent` function, as previously described, to open a component without calling the `FindNextComponent` function.

Note that your application may maintain several connections to a single component, or it may have connections to several components at the same time.

```
FUNCTION OpenComponent (aComponent: Component): ComponentInstance;
```

`aComponent`
: A component identifier that specifies the component to open. Your application obtains this identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.

DESCRIPTION

The `OpenComponent` function returns a component instance. The returned component instance identifies your application's connection to the component. You must supply this component instance whenever you call the functions provided by the component. When you close the component, you must also supply this component instance to the `CloseComponent` function.

If it cannot open the specfied component, the `OpenComponent` function returns a function result of `NIL`.

SEE ALSO

For examples of opening a specific component by using the `FindNextComponent` and `OpenComponent` functions, see Listing 6-1 on page 6-9 and Listing 6-2 on page 6-10, respectively. For a description of the `FindNextComponent` function, see page 6-42.

CloseComponent

The `CloseComponent` function terminates your application's access to the services provided by a component. Your application specifies the connection to be closed with the component instance returned by the `OpenComponent` or `OpenDefaultComponent` function.

```
FUNCTION CloseComponent
            (aComponentInstance: ComponentInstance): OSErr;
```

`aComponentInstance`
: A component instance that specifies the connection to close. Your application obtains the component instance from the `OpenComponent` function or the `OpenDefaultComponent` function.

DESCRIPTION

The `CloseComponent` function closes only a single connection. If your application has several connections to a single component, you must call the `CloseComponent` function once for each connection.

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	No component with this component identifier

SEE ALSO

For a description of the `OpenDefaultComponent` and `OpenComponent` functions, see page 6-45 and page 6-46, respectively.

Getting Information About Components

Your application can get the registration information for any component using the `GetComponentInfo` function. You can use the `GetComponentIconSuite` function to get a handle to the component's icon suite, if any.

In addition, for components to which your application already has a connection, your application can obtain the component's version number and also determine whether the component supports a particular request by using the `GetComponentVersion` and `ComponentFunctionImplemented` functions.

GetComponentInfo

The `GetComponentInfo` function returns all of the registration information for a component. Your application specifies the component with a component identifier returned by the `FindNextComponent` function. The `GetComponentInfo` function returns information about the component in a component description record. The `GetComponentInfo` function also returns the component's name, information string, and icon. (To get a handle to the component's icon suite, if it provides one, use the `GetComponentIconSuite` function.)

A component provides this registration information when it is registered with the Component Manager.

```
FUNCTION GetComponentInfo (aComponent: Component;
                           VAR cd: ComponentDescription;
                           componentName: Handle;
                           componentInfo: Handle;
                           componentIcon: Handle): OSErr;
```

`aComponent`
: A component identifier that specifies the component for the operation. Your application obtains a component identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.

  You may supply a component instance rather than a component identifier to this function. (If you do so, you must coerce the data type appropriately.) Your application can obtain a component instance from the `OpenComponent` function or the `OpenDefaultComponent` function.

`cd`
: A component description record. The `GetComponentInfo` function returns information about the specified component in a component description record.

`componentName`
: An existing handle that is to receive the component's name. If the component does not have a name, the `GetComponentInfo` function returns an empty handle. Set this field to `NIL` if you do not want to receive the component's name.

`componentInfo`
: An existing handle that is to receive the component's information string. If the component does not have an information string, the `GetComponentInfo` function returns an empty handle. Set this field to `NIL` if you do not want to receive the component's information string.

`componentIcon`
: An existing handle that is to receive the component's icon. If the component does not have an icon, the `GetComponentInfo` function returns an empty handle. Set this field to `NIL` if you do not want to receive the component's icon.

DESCRIPTION

The `GetComponentInfo` function returns information about the specified component in the `cd`, `componentName`, `componentInfo`, and `componentIcon` parameters.

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	No component with this component identifier

SEE ALSO

For information on the component description record, see page 6-37. For information on the `FindNextComponent` function, see page 6-42. For information on registering components, see “Registering Components” beginning on page 6-57.

For an example of the use of the `GetComponentInfo` function, see Listing 6-3 on page 6-10.

GetComponentIconSuite

The `GetComponentIconSuite` function returns a handle to the component’s icon suite (if it provides one).

```
FUNCTION GetComponentIconSuite (aComponent: Component;
                                VAR iconSuite: Handle): OSErr;
```

`aComponent`
: A component identifier that specifies the component for the operation. Your application obtains a component identifier from the `FindNextComponent` function. If your application registers a component, it can also obtain a component identifier from the `RegisterComponent` or `RegisterComponentResource` function.

`iconSuite`
: `GetComponentIconSuite` returns, in this parameter, a handle to the component’s icon suite, if any. If the component has not provided an icon suite, `GetComponentIconSuite` returns `NIL` in this parameter.

DESCRIPTION

The `GetComponentIconSuite` function returns a handle to the component’s icon suite. A component provides to the Component Manager the resource ID of its icon family in the optional extensions to the component resource. Your application is responsible for disposing of the returned icon suite handle.

SPECIAL CONSIDERATIONS

The `GetComponentIconSuite` function is available only in version 3 of the Component Manager.

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	No component with this component identifier

SEE ALSO

For information about icon suites and icon families, see the chapter "Icon Utilities" in this book.

GetComponentVersion

The `GetComponentVersion` function returns a component's version number.

```
FUNCTION GetComponentVersion (ci: ComponentInstance): LongInt;
```

`ci` The component instance from which you want to retrieve version information. Your application obtains the component instance from the `OpenDefaultComponent` or `OpenComponent` function.

DESCRIPTION

The `GetComponentVersion` function returns a long integer containing the version number of the component you specify. The high-order 16 bits represent the major version, and the low-order 16 bits represent the minor version. The major version specifies the component specification level; the minor version specifies a particular implementation's version number.

ComponentFunctionImplemented

The `ComponentFunctionImplemented` function allows you to determine whether a component supports a specified request. Your application can use this function to determine a component's capabilities.

```
FUNCTION ComponentFunctionImplemented (ci: ComponentInstance;
                                        ftnNumber: Integer)
                                        : LongInt;
```

`ci` A component instance that specifies the connection for this operation. Your application obtains the component instance from the `OpenDefaultComponent` or `OpenComponent` function.

`ftnNumber` A request code value. See *Inside Macintosh: QuickTime Components* for information about the request codes supported by the components supplied by Apple with QuickTime. For other components, see the documentation supplied with the component for request code values.

DESCRIPTION

The `ComponentFunctionImplemented` function returns a long integer indicating whether the component supports the specified request. You can interpret this long integer as if it were a Boolean value. If the returned value is `TRUE`, the component supports the specified request. If the returned value is `FALSE`, the component does not support the request.

Retrieving Component Errors

The Component Manager provides a routine that allows your application to retrieve the last error code that was generated by a component instance. Some component routines return error information as their function result. Other component routines set an error code that your application can retrieve using the `GetComponentInstanceError` function. Refer to the documentation supplied with the component for information on how that particular component handles errors.

GetComponentInstanceError

The `GetComponentInstanceError` function returns the last error generated by a specific connection to a component.

```
FUNCTION GetComponentInstanceError
                    (aComponentInstance: ComponentInstance): OSErr;
```

`aComponentInstance`
A component instance that specifies the connection from which you want error information. Your application obtains the component instance from the `OpenDefaultComponent` or `OpenComponent` function.

DESCRIPTION

Once you have retrieved an error code, the Component Manager clears the error code for the connection. If you want to retain that error value, you should save it in your application's local storage.

RESULT CODES

noErr	0	No error
invalidComponentID	–3000	No component with this component identifier

Data Structures for Components

This section describes the format and content of the data structures used by components.

Components, and applications that register components, use the component description record to identify a component. A component resource incorporates the information in a component description record and also includes other information. If you are developing a component or an application that registers components, you must be familiar with both the component description record and component resource; see “Resources” beginning on page 6-80 for a description of the component resource.

The Component Manager passes information about a request to your component in a component parameters record.

The Component Description Record

The component description record identifies the characteristics of a component, including the type of services offered by the component and the manufacturer of the component.

Components use component description records to identify themselves to the Component Manager. If your component is stored in a component resource, the information in the component description record must be part of that resource (see the description of the component resource, on page 6-80). If you have developed an application that registers your component, that application must supply a component description record to the `RegisterComponent` function (see “Registering Components” on page 6-57 for information about registering components).

The `ComponentDescription` data type defines the component description record. Note that the valid values of fields in the component description record are determined by the component type specification. For example, all image compressor components must use the `componentSubType` field to specify the compression algorithm used by the compressor.

```
TYPE  ComponentDescription =
      RECORD
         componentType:        OSType;      {type}
         componentSubType:     OSType;      {subtype}
         componentManufacturer:             {manufacturer}
                               OSType;
         componentFlags:       LongInt;     {control flags}
         componentFlagsMask:   LongInt;     {reserved}
      END;
```

Field descriptions

`componentType`

A four-character code that identifies the type of component. All components of a particular type must support a common set of interface routines. For example, drawing components all have a component type of `'draw'`.

Your component must support all of the standard routines for the component type specified by this field. Type codes with all lowercase characters are reserved for definition by Apple. See *Inside Macintosh: QuickTime Components* for information about the QuickTime components supplied by Apple. You can define your own component type code as long as you register it with Apple's Component Registry Group.

`componentSubType`

A four-character code that identifies the subtype of the component. Different subtypes of a component type may support additional features or provide interfaces that extend beyond the standard routines for a given component. For example, the subtype of a drawing component indicates the type of object the component draws. Drawing components that draw ovals have a subtype of `'oval'`.

Your component may use this field to indicate more specific information about the capabilities of the component. There are no restrictions on the content you assign to this field. If no additional information is appropriate for your component type, you may set the `componentSubType` field to 0.

`componentManufacturer`

A four-character code that identifies the manufacturer of the component. This field allows for further differentiation between individual components. For example, components made by a specific manufacturer may support an extended feature set. Components provided by Apple use a manufacturer value of `'appl'`.

Your component uses this field to indicate the manufacturer of the component. You obtain your manufacturer code, which can be the same as your application signature, from Apple's Component Registry Group.

`componentFlags`

A 32-bit field that provides additional information about a particular component.

The high-order 8 bits are reserved for definition by the Component Manager and provide information about the component. The following bits are currently defined:

```
CONST
 cmpWantsRegisterMessage = $80000000;
 cmpFastDispatch         = $40000000;
```

The setting of the `cmpWantsRegisterMessage` bit determines whether the Component Manager calls this component during registration. Set this bit to 1 if your component should be called when it is registered; otherwise, set this bit to 0. If you want to automatically dispatch requests to your component to the appropriate routine that handles the request (rather than your component calling `CallComponentFunction` or `CallComponentFunctionWithStorage`), set the `cmpFastDispatch` bit. If you set this bit, you must write your component's entry point in assembly language. If you set this bit, the Component Manager calls your component's entry point with the call's parameters, the handle to that instance's storage, and the caller's return address already on the stack. The Component Manager passes the request code in register D0 and passes the stack location of the instance's storage in register A0. Your component can then use the request code in register D0 to directly dispatch the request itself (for example, by using this value as an index into a table of function addresses). Be sure to note that the standard request codes have negative values. Also note that the function parameter that the caller uses to specify the component instance instead contains a handle to the instance's storage. When the component function completes, control returns to the calling application.

For more information about component registration and initialization, see "Responding to the Register Request" on page 6-23.

The low-order 24 bits are specific to each component type. You can use these flags to indicate any special capabilities or features of your component. Your component may use all 24 bits, as appropriate to its component type. You must set all unused bits to 0.

`componentFlagsMask`

Reserved. (However, note that applications can use this field when performing search operations, as described on page 6-39.)

Your component must set the `componentFlagsMask` field in its component description record to 0.

The Component Parameters Record

The Component Manager uses the component parameters record to pass information to your component about a request from an application. The information in this record completely defines the request. Your component services the request as appropriate.

The `ComponentParameters` data type defines the component parameters record.

```
ComponentParameters =
      PACKED RECORD
         flags:      Char;                     {reserved}
         paramSize:  Char;                     {size of parameters}
         what:       Integer;                  {request code}
         params:     ARRAY[0..0] OF LongInt; {actual parameters}
      END;
```

Field descriptions

`flags` Reserved for use by Apple.

`paramSize` Specifies the number of bytes of parameter data for this request. The actual parameters are stored in the `params` field.

`what` Specifies the type of request. Component designers define the meaning of positive values and assign them to requests that are supported by components of a given type. Negative values are reserved for definition by Apple. Apple has defined these request codes:

```
CONST
   kComponentOpenSelect       = -1; {required}
   kComponentCloseSelect      = -2; {required}
   kComponentCanDoSelect      = -3; {required}
   kComponentVersionSelect    = -4; {required}
   kComponentRegisterSelect   = -5; {optional}
   kComponentTargetSelect     = -6; {optional}
   kComponentUnregisterSelect = -7; {optional}
```

`params` An array that contains the parameters specified by the application that called your component.
You can use the `CallComponentFunction` or `CallComponentFunctionWithStorage` routine to convert this array into a Pascal-style invocation of a subroutine in your component.

For information on how your component responds to requests, see "Handling Requests for Service" beginning on page 6-18.

Routines for Components

This section describes the Component Manager routines that are used by components. It also discusses routines a component or application can use to register a component. This section first describes the routines for registering components then describes the routines that allow your component to

- extract the parameters from a component parameters record and invoke a subroutine of your component with these parameters
- manage open connections
- associate storage with a specific connection
- pass error information to the Component Manager for later use by the calling application
- store and retrieve your component's reference constant
- open and close its resource file
- call other components
- capture other components
- target a component instance

Note that version 3 and above of the Component Manager supports automatic version control, the unregister request, and icon families. You should test the version number before using any of these features. You can use the `Gestalt` function with the `gestaltComponentMgr` selector to do this. When you specify this selector, `Gestalt` returns in the `response` parameter a 32-bit value indicating the version of the Component Manager that is installed.

If you are developing an application that uses components but does not register them, you do not have to read this material, though it may be interesting to you. For a discussion of the Component Manager routines that support applications that use components, see "Routines for Applications" beginning on page 6-41.

If you are developing an application that registers components, you should read the next section, "Registering Components." You may also find the other topics in this section interesting.

If you are developing a component, you should read this entire section. For more information about creating components, see "Creating Components" beginning on page 6-13.

Several of the routines discussed in this section use the component parameters record. For a complete description of that structure, see "Data Structures for Components" beginning on page 6-52. For information on the distinction between component identifiers and component instances, see page 6-40.

Note

Any of the routines discussed in this section that require a component identifier also accept a component instance. Similarly, you can supply a component identifier to any routine that requires a component instance (except for the `DelegateComponentCall` function). If you do this, you must always coerce the data type appropriately. For more information, see “Component Identifiers and Component Instances” on page 6-40. ◆

Registering Components

Before a component can be used by an application, the component must be registered with the Component Manager. The Component Manager automatically registers component resources stored in files with file types of `'thng'` that are stored in the Extensions folder (for information about the content of component resources, see “Resources” beginning on page 6-80).

Alternatively, you can use either the `RegisterComponent` function or the `RegisterComponentResource` function to register components. Both applications and components can use these routines to register components.

Furthermore, you can use the `RegisterComponentResourceFile` function to register all components specified in a given resource file.

Once you have registered your component, applications can find the component and retrieve information about it using the Component Manager routines described earlier in this chapter in “Routines for Applications” beginning on page 6-41.

Finally, you can use the `UnregisterComponent` function to remove a component from the registration list.

Note

When an application quits, the Component Manager automatically closes any component connections to that application. In addition, if the application has registered components that reside in its heap space, the Component Manager automatically unregisters those components. ◆

RegisterComponent

The `RegisterComponent` function makes a component available for use by applications (or other clients). Once the Component Manager has registered a component, applications can find and open the component using the standard Component Manager routines. To register a component, you provide information identifying the component and its capabilities. The Component Manager returns a component identifier that uniquely identifies the component to the system.

Components you register with the `RegisterComponent` function must be in memory when you call this function. If you want to register a component that is stored in the resource fork of a file, use the `RegisterComponentResource` function. Use the `RegisterComponentResourceFile` function to register all components in the resource fork of a file.

Note that a component residing in your application heap remains registered until your application unregisters it or quits. A component residing in the system heap and registered by your application remains registered until your application unregisters it or until the computer is shut down.

```
FUNCTION RegisterComponent (cd: ComponentDescription;
                           componentEntryPoint: ComponentRoutine;
                           global: Integer;
                           componentName: Handle;
                           componentInfo: Handle;
                           componentIcon: Handle): Component;
```

`cd` A component description record that describes the component to be registered. You must correctly fill in the fields of this record before calling the `RegisterComponent` function. When applications search for components using the `FindNextComponent` function, the Component Manager compares the attributes you specify here with those specified by the application. If the attributes match, the Component Manager returns the component identifier to the application.

`componentEntryPoint`
The address of the main entry point of the component you are registering. The routine referred to by this parameter receives all requests for the component.

`global` A set of flags that control the scope of component registration. You can use these flags to specify a value for the `global` parameter:

`registerCmpGlobal = 1;`
Specify this flag to indicate that this component should be made available to other applications and clients as well as the one performing the registration. If you do not specify this flag, the component is available for use only by the registering application or component (that is, the component is local to the A5 world of the registering program).

`registerCmpNoDuplicates = 2;`
Specify this flag to indicate that if a component with identical characteristics to the one being registered already exists, then the new one should not be registered (`RegisterComponent` returns 0 in this situation). If you do not specify this flag, the component is registered even if a component with identical characteristics to the one being registered already exists.

`registerCompAfter = 4;`
: Specify this flag to indicate that this component should be registered after all other components with the same component type. Usually components are registered before others with identical descriptions; specifying this flag overrides that behavior.

`componentName`
: A handle to the component's name. Set this parameter to `NIL` if you do not want to assign a name to the component.

`componentInfo`
: A handle to the component's information string. Set this parameter to `NIL` if you do not want to assign an information string to the component.

`componentIcon`
: A handle to the component's icon (a 32-by-32 pixel black-and-white icon). Set this parameter to `NIL` if you do not want to supply an icon for this component. Note that this icon is not used by the Finder; you supply an icon only so that other components or applications can display your component's icon if needed.

DESCRIPTION

The `RegisterComponent` function registers the specified component, recording the information specified in the `cd`, `componentName`, `componentInfo`, and `componentIcon` parameters. The function returns the component identifier assigned to the component by the Component Manager. If it cannot register the component, the `RegisterComponent` function returns a function result of `NIL`.

SEE ALSO

For a complete description of the component description record, see "Data Structures for Components" beginning on page 6-52.

RegisterComponentResource

The `RegisterComponentResource` function makes a component available for use by applications (or other clients). Once the Component Manager has registered a component, applications can find and open the component using the standard Component Manager routines. You provide information identifying the component and specifying its capabilities. The Component Manager returns a component identifier that uniquely identifies the component to the system.

Components you register with the `RegisterComponentResource` function must be stored in a resource file as a component resource (see "The Component Resource" beginning on page 6-80 for a description of the format and content of component resources). If you want to register a component that is in memory, use the `RegisterComponent` function.

The `RegisterComponentResource` function does not actually load the code specified by the component resource into memory. Rather, the Component Manager loads the component code the first time an application opens the component. If the code is not in the same file as the component resource or if the Component Manager cannot find the file, the open request fails.

Note that a component registered locally by your application remains registered until your application unregisters it or quits. A component registered globally by your application remains registered until your application unregisters it or until the computer is shut down.

```
FUNCTION RegisterComponentResource (cr: ComponentResourceHandle;
                                    global: Integer): Component;
```

`cr` A handle to a component resource that describes the component to be registered. The component resource contains all the information required to register the component.

`global` A set of flags that controls the scope of component registration. You can use these flags to specify a value for the `global` parameter:

`registerCmpGlobal = 1;`
Specify this flag to indicate that this component should be made available to other applications and clients as well as the one performing the registration. If you do not specify this flag, the component is available for use only by the registering application or component (that is, the component is local to the A5 world of the registering program).

`registerCmpNoDuplicates = 2;`
Specify this flag to indicate that if a component with identical characteristics to the one being registered already exists, then the new one should not be registered (`RegisterComponentResource` returns 0 in this situation). If you do not specify this flag, the component is registered even if a component with identical characteristics to the one being registered already exists.

`registerCompAfter = 4;`
Specify this flag to indicate that this component should be registered after all other components with the same component type. Usually components are registered before others with identical descriptions; specifying this flag overrides that behavior.

DESCRIPTION

The `RegisterComponentResource` function returns the component identifier assigned to the component by the Component Manager. If the `RegisterComponentResource` function could not register the component, it returns a function result of `NIL`.

SEE ALSO

For a description of the format and content of component resources, see “Resources” beginning on page 6-80.

RegisterComponentResourceFile

The `RegisterComponentResourceFile` function registers all component resources in the given resource file according to the flags specified in the `global` parameter.

```
FUNCTION RegisterComponentResourceFile (resRefNum: integer;
                                        global: integer): LongInt;
```

`resRefNum` The reference number of the resource file containing the components to register.

`global` A set of flags that control the scope of the registration of the components in the resource file specified in the `resRefNum` parameter. You can use these flags to specify a value for the `global` parameter:

`registerCmpGlobal = 1;`
Specify this flag to indicate that each component in the resource file should be made available to other applications and clients as well as the one performing the registration. If you do not specify this flag, each component is available for use only by the registering application or component (that is, the component is local to the A5 world of the registering program).

`registerCmpNoDuplicates = 2;`
Specify this flag to indicate that if a component with identical characteristics to the one being registered already exists, then the new one should not be registered (`RegisterComponentResourceFile` returns 0 in this situation). If you do not specify this flag, the component is registered even if a component with identical characteristics to the one being registered already exists.

`registerCompAfter = 4;`
Specify this flag to indicate that as `RegisterComponentResourceFile` registers a component, it should register the component after all other components with the same component type. Usually components are registered before others with identical descriptions; specifying this flag overrides that behavior.

DESCRIPTION

The `RegisterComponentResourceFile` function registers components in a resource file. If the `RegisterComponentResourceFile` function successfully registers all components in the specified resource file, `RegisterComponentResourceFile` returns

a function result that indicates the number of components registered. If the `RegisterComponentResourceFile` function could not register one or more of the components in the resource file or if the specified file reference number is invalid, it returns a negative function result.

SEE ALSO

For a description of the format and content of component resources, see "Resources" beginning on page 6-80.

UnregisterComponent

The `UnregisterComponent` function removes a component from the Component Manager's registration list. Most components are registered at startup and remain registered until the computer is shut down. However, you may want to provide some services temporarily. In that case you dispose of the component that provides the temporary service by using this function.

```
FUNCTION UnregisterComponent (aComponent: Component): OSErr;
```

`aComponent`
: A component identifier that specifies the component to be removed. Applications that register components may obtain this identifier from the `RegisterComponent` or `RegisterComponentResource` functions.

DESCRIPTION

The `UnregisterComponent` function removes the component with the specified component identifier from the list of available components. The component to be removed from the registration list must not be in use by any applications or components. If there are open connections to the component, the `UnregisterComponent` function returns a negative result code.

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	No component with this component identifier
`validInstancesExist`	–3001	This component has open connections

SEE ALSO

If you provide a component that supports the unregister request, see "Responding to the Register Request" on page 6-23 for more information.

Dispatching to Component Routines

This section discusses routines that simplify the process of calling subroutines within your component.

When an application requests service from your component, your component receives a component parameters record containing the information for that request. That component parameters record contains the parameters that the application provided when it called your component. Your component can use this record to access the parameters directly. Alternatively, you can use the routines described in this section to extract those parameters and pass them to a subroutine of your component. By taking advantage of these routines, you can simplify the structure of your component code. For more information about the interface between the Component Manager and your component, see "Creating Components" beginning on page 6-13.

Use the `CallComponentFunction` function to call a component subroutine without providing it access to global data for that connection. Use the `CallComponentFunctionWithStorage` function to call a component subroutine and to pass it a handle to the memory that stores the global data for that connection.

CallComponentFunction

The `CallComponentFunction` function invokes a specified function of your component with the parameters originally provided by the application that called your component. You pass these parameters by specifying the same component parameters record passed to your component's main entry point.

```
FUNCTION CallComponentFunction (params: ComponentParameters;
                                func: ComponentFunction): LongInt;
```

`params` The component parameters record that your component received from the Component Manager.

`func` The address of the function that is to handle the request. The Component Manager calls the routine referred to by the `func` parameter as a Pascal function with the parameters that were originally provided by the application. The routine referred to by this parameter must return a function result of type `ComponentResult` (a long integer) indicating the success or failure of the operation.

DESCRIPTION

`CallComponentFunction` returns the value that is returned by the routine referred to by the `func` parameter. Your component should use this value to set the current error for this connection.

SPECIAL CONSIDERATIONS

If your component subroutine does not need global data, your component should use `CallComponentFunction`. If your component subroutine requires memory in which to store global data for the component, your component must use `CallComponentFunctionWithStorage`, which is described next.

SEE ALSO

For an example that uses `CallComponentFunction`, see Listing 6-5 on page 6-16. You can use the `SetComponentInstanceError` procedure, described on page 6-69, to set the current error.

CallComponentFunctionWithStorage

The `CallComponentFunctionWithStorage` function invokes a specified function of your component with the parameters originally provided by the application that called your component. You pass these parameters by specifying the same component parameters record that was received by your component's main entry point. The `CallComponentFunctionWithStorage` function also provides a handle to the memory associated with the current connection.

```
FUNCTION CallComponentFunctionWithStorage
               (storage: Handle; params: ComponentParameters;
                func: ComponentFunction): LongInt;
```

`storage` A handle to the memory associated with the current connection. The Component Manager provides this handle to your component along with the request.

`params` The component parameters record that your component received from the Component Manager.

`func` The address of the function that is to handle the request. The Component Manager calls the routine referred to by the `func` parameter as a Pascal function with the parameters that were originally provided by the application. These parameters are preceded by a handle to the memory associated with the current connection. The routine referred to by the `func` parameter must return a function result of type `ComponentResult` (a long integer) indicating the success or failure of the operation.

DESCRIPTION

The `CallComponentFunctionWithStorage` function returns the value that is returned by the function referred to by the `func` parameter. Your component should use this value to set the current error for this connection.

SPECIAL CONSIDERATIONS

`CallComponentFunctionWithStorage` takes as a parameter a handle to the memory associated with the connection, so subroutines of a component that don't need global data should use the `CallComponentFunction` routine described in the previous section instead.

If your component subroutine requires a handle to the memory associated with the connection, you must use `CallComponentFunctionWithStorage`. You allocate the memory for a given connection each time your component is opened. You inform the Component Manager that a connection has memory associated with it by calling the `SetComponentInstanceStorage` procedure.

SEE ALSO

For an example that uses `CallComponentFunctionWithStorage`, see Listing 6-5 on page 6-16. Use the `SetComponentInstanceError` procedure, described on page 6-69, to set the current error for a connection. A description of the `SetComponentInstanceStorage` procedure is given next.

Managing Component Connections

The Component Manager provides a number of routines that help your component manage the connections it maintains with its client applications and components.

Use the `SetComponentInstanceStorage` procedure to inform the Component Manager of the memory your component is using to maintain global data for a connection. Whenever the client application issues a request to the connection, the Component Manager provides to your component the handle to the allocated memory for that connection along with the parameters for the request. You can also use the `GetComponentInstanceStorage` function to retrieve a handle to the storage for a connection.

Use the `CountComponentInstances` function to count all the connections that are currently maintained by your component. This routine is similar to the `CountComponents` routine that the Component Manager provides to client applications and components.

Use the `SetComponentInstanceA5` procedure to set the A5 world for a connection. Once you set the A5 world for a connection, the Component Manager automatically switches the contents of the A5 register when your component receives a request for that connection. When your component returns to the Component Manager, the Component Manager restores the A5 register. Your component can use the `GetComponentInstanceA5` function to retrieve the A5 world for a connection.

SetComponentInstanceStorage

When an application or component opens a connection to your component, the Component Manager sends your component an open request. In response to this open request, your component should set up an environment to service the connection. Typically, your component should allocate some memory for the connection. Your component can then use that memory to maintain state information appropriate to the connection.

The `SetComponentInstanceStorage` procedure allows your component to pass a handle to this memory to the Component Manager. The Component Manager then provides this handle to your component each time the client application requests service from this connection.

```
PROCEDURE SetComponentInstanceStorage
      (aComponentInstance: ComponentInstance; theStorage: Handle);
```

`aComponentInstance`
: The connection to associate with the allocated memory. The Component Manager provides a component instance to your component when the connection is opened.

`theStorage`
: A handle to the memory that your component has allocated for the connection. Your component must allocate this memory in the current heap. The Component Manager saves this handle and provides it to your component, along with other parameters, in subsequent requests to this connection.

DESCRIPTION

The `SetComponentInstanceStorage` procedure associates the handle passed in the parameter `theStorage` with the connection specified by the `aComponentInstance` parameter. Your component should dispose of any allocated memory for the connection only in response to the close request.

SPECIAL CONSIDERATIONS

Note that whenever an open request fails, the Component Manager always issues the close request. Furthermore, the value stored with `SetComponentInstanceStorage` is always passed to the close request, so it must be valid or `NIL`. If the open request tries to dispose of its allocated memory before returning, it should call `SetComponentInstanceStorage` again with a `NIL` handle to keep the Component Manager from passing an invalid handle to the close request.

SEE ALSO

For an example that allocates memory in response to an open request, see Listing 6-6 on page 6-20.

GetComponentInstanceStorage

The `GetComponentInstanceStorage` function allows your component to retrieve a handle to the memory associated with a connection. Your component tells the Component Manager about this memory by calling the `SetComponentInstanceStorage` procedure. Typically, your component does not need to use this function, because the Component Manager provides this handle to your component each time the client application requests service from this connection.

```
FUNCTION GetComponentInstanceStorage
                    (aComponentInstance: ComponentInstance): Handle;
```

`aComponentInstance`
: The connection for which to retrieve the associated memory. The Component Manager provides a component instance to your component when the connection is opened.

DESCRIPTION

The `GetComponentInstanceStorage` function returns a handle to the memory associated with the specified connection.

CountComponentInstances

The `CountComponentInstances` function allows you to determine the number of open connections being managed by a specified component. This function can be useful if you want to restrict the number of connections for your component or if your component needs to perform special processing based on the number of open connections.

```
FUNCTION CountComponentInstances (aComponent: Component): LongInt;
```

`aComponent`
: The component for which you want a count of open connections. You can use the component instance that your component received in its open request to identify your component.

DESCRIPTION

The `CountComponentInstances` function returns the number of open connections for the specified component.

SetComponentInstanceA5

The `SetComponentInstanceA5` procedure allows your component to set the A5 world for a connection.

```
PROCEDURE SetComponentInstanceA5
        (aComponentInstance: ComponentInstance; theA5: LongInt);
```

`aComponentInstance`
: The connection for which to set the A5 world. The Component Manager provides a component instance to your component when the connection is opened.

`theA5`
: The value of the A5 register for the connection. The Component Manager sets the A5 register to this value automatically, and it restores the previous A5 value when your component returns to the Component Manager.

DESCRIPTION

The `SetComponentInstanceA5` procedure sets the A5 world for the specified component instance. Once you set the A5 world for a connection, the Component Manager automatically switches the contents of the A5 register when your component receives a request over that connection. When your component returns to the Component Manager, the Component Manager restores your client's A5 value.

If your component has been registered globally and you have not set an A5 value, the A5 register is set to 0. In this case you should set the A5 world of your component instance to your client's A5 value by using `SetComponentInstanceA5`.

In general, your component uses this procedure only if it is registered globally; in this case, it typically calls `SetComponentInstanceA5` when processing the open request for a new connection.

GetComponentInstanceA5

You can use the `GetComponentInstanceA5` function to retrieve the value of the A5 register for a specified connection. Your component sets the A5 register by calling the `SetComponentInstanceA5` function, as previously described. The Component Manager then sets the A5 register for your component each time the client requests

service on this connection. If your component has been registered globally and you have not set an A5 value, the A5 register is set to 0. In this case you should use your client's A5 value.

```
FUNCTION GetComponentInstanceA5
         (aComponentInstance: ComponentInstance): LongInt;
```

`aComponentInstance`
: The connection for which to retrieve the A5 value. The Component Manager provides a component instance to your component when the connection is opened.

DESCRIPTION

The `GetComponentInstanceA5` function returns the value of the A5 register for the connection.

Setting Component Errors

The Component Manager maintains error state information for all currently active components. In general, your component returns error information in its function result; a nonzero function result indicates an error occurred, and a function result of 0 indicates the request was successful. However, some requests require that your component return other information as its function result. In these cases, your component can use the `SetComponentInstanceError` procedure to report its latest error state to the Component Manager. You can also use this procedure at any time during your component's execution to report an error

SetComponentInstanceError

Although your component usually returns error information as its function result, your component can choose to use the `SetComponentInstanceError` procedure to pass error information to the Component Manager. The Component Manager uses this error information to set the current error value for the appropriate connection. Applications can then retrieve this error information by calling the `GetComponentInstanceError` function. The documentation for your component should specify how the component indicates errors.

```
PROCEDURE SetComponentInstanceError
         (aComponentInstance: ComponentInstance; theError: OSErr);
```

`aComponentInstance`
: A component instance that specifes the connection for which to set the error. The Component Manager provides a component instance to your component when the connection is opened. The Component Manager also provides a component instance to your component as the first parameter in the `params` field of the parameters record.

`theError`
: The new value for the current error. The Component Manager uses this value to set the current error for the connection specified by the `aComponentInstance` parameter.

DESCRIPTION

The `SetComponentInstanceError` procedure sets the error associated with the specified component instance to the value specified by the parameter `theError`.

SEE ALSO

For a description of the `GetComponentInstanceError` function, see page 6-51.

Working With Component Reference Constants

The Component Manager provides routines that manage access to the reference constants that are associated with components. There is one reference constant for each component, regardless of the number of connections to that component. When your component is registered, the Component Manager sets this reference constant to 0.

The reference constant is a 4-byte value that your component can use in any way you decide. For example, you might use the reference constant to store the address of a data structure that is shared by all connections maintained by your component. You should allocate shared structures in the system heap. Your component should deallocate the structure when its last connection is closed or when it is unregistered.

Use the `SetComponentRefcon` procedure to set the value of the reference constant for your component. Use the `GetComponentRefcon` function to retrieve the value of the reference constant.

SetComponentRefcon

You can use the `SetComponentRefcon` procedure to set the reference constant for your component.

```
PROCEDURE SetComponentRefcon (aComponent: Component;
                              theRefcon: LongInt);
```

`aComponent`
A component identifier that specifies the component whose reference constant you wish to set.

`theRefCon` The reference constant value that you want to set for your component.

DESCRIPTION

The `SetComponentRefcon` procedure sets the value of the reference constant for your component. Your component can later retrieve the reference constant using the `GetComponentRefcon` function, described next.

GetComponentRefcon

The `GetComponentRefcon` function retrieves the value of the reference constant for your component.

```
FUNCTION GetComponentRefcon (aComponent: Component): LongInt;
```

`aComponent`
A component identifier that specifies the component whose reference constant you wish to get.

DESCRIPTION

The `GetComponentRefcon` function returns a long integer containing the reference constant for the specified component.

Accessing a Component's Resource File

If you store your component in a component resource and register your component using the `RegisterComponentResource` function or `RegisterComponentResourceFile` function, or if the Component Manager automatically registers your component, the Component Manager allows your component to gain access to its resource file. You can store read-only data for your component in its resource file. For example, the resource file may contain the color icon for the component, static data needed to initialize private storage, or any other data that may be useful to the component. Note that there is only one resource file associated with a component.

If you store your component in a component resource but register the component with the `RegisterComponent` function, rather than with the `RegisterComponentResource` or `RegisterComponentResourceFile` function, your component cannot access its resource file with the routines described in this section.

The routines described in this section allow your component to gain access to its resource file. These routines provide read-only access to the data in the resource file. If your component opens its resource file, it must close the file before returning to the calling application.

Use the `OpenComponentResFile` function to open your component's resource file. Use the `CloseComponentResFile` function to close the resource file before returning to the calling application.

OpenComponentResFile

The `OpenComponentResFile` function allows your component to gain access to its resource file. This function opens the resource file with read permission and returns a reference number that your component can use to read data from the file. The Component Manager adds the resource file to the current resource chain. Your component must close the resource file with the `CloseComponentResFile` function before returning to the calling application.

Your component can use `FSpOpenResFile` or equivalent Resource Manager routines to open other resource files, but you must use `OpenComponentResFile` to open your component's resource file.

```
FUNCTION OpenComponentResFile (aComponent: Component): Integer;
```

`aComponent`
: A component identifier that specifies the component whose resource file you wish to open. Applications that register components may obtain this identifier from the `RegisterComponentResource` function.

DESCRIPTION

The `OpenComponentResFile` function returns a reference number for the appropriate resource file. This function returns 0 or a negative number if the specified component does not have an associated resource file or if the Component Manager cannot open the resource file.

Note that when working with resources, your component should always first save the current resource file, perform any resource operations, then restore the current resource file to its previous value before returning.

CloseComponentResFile

This function closes the resource file that your component opened previously with the `OpenComponentResFile` function.

```
FUNCTION CloseComponentResFile (refnum: Integer): OSErr;
```

`refnum` The reference number that identifies the resource file to be closed. Your component obtains this value from the `OpenComponentResFile` function.

DESCRIPTION

The `CloseComponentResFile` function closes the specified resource file. Your component must close any open resource files before returning to the calling application.

RESULT CODES

noErr	0	No error
resFNotFound	–193	Resource file not found

Calling Other Components

The Component Manager provides two techniques that allow a component to call other components. First, your component may invoke the services of another component using the standard mechanisms also used by applications. The Component Manager then passes the requests to the appropriate component, and your component receives the results of those requests.

Second, your component may supplement its capabilities by using the services of another component to directly satisfy application requests. The Component Manager provides the `DelegateComponentCall` function, which allows your component to pass a request to a specified component. For example, you might want to create two similar components that provide different levels of service to applications. Rather than completely implementing both components, you could design one to rely on the capabilities of the other. In this manner, you have to implement only that portion of the more capable component that provides additional services.

DelegateComponentCall

The `DelegateComponentCall` function provides an efficient mechanism for passing on requests to a specified component. Your component must open a connection to the component to which the requests are to be passed. Your component must close that connection when it has finished using the services of the other component.

Note
The `DelegateComponentCall` function does not accept a component identifier in place of a component instance. In addition, your component should never use the `DelegateComponentCall` function with open or close requests from the Component Manager—always use the `OpenComponent` and `CloseComponent` functions to manage connections with other components. ◆

```
FUNCTION DelegateComponentCall
                         (originalParams: ComponentParameters;
                          ci: ComponentInstance): LongInt;
```

`originalParams`
The component parameters record provided to your component by the Component Manager.

`ci` The component instance that is to process the request. The Component Manager provides a component instance to your component when it opens a connection to another component with the `OpenComponent` or `OpenDefaultComponent` function. You must specify a component instance; this function does not accept a component identifier.

DESCRIPTION

The `DelegateComponentCall` function calls the component instance specified by the `ci` parameter, and passes it the specified component parameters record. `DelegateComponentCall` returns a long integer containing the component result returned by the specified component.

SEE ALSO

See "The Component Parameters Record" on page 6-54 for a description of the component parameters record. See page 6-45, page 6-46, and page 6-47, respectively, for information on the `OpenDefaultComponent`, `OpenComponent`, and `CloseComponent` functions.

See Listing 6-16 on page 6-36 for an example of the use of the `DelegateComponentCall` function.

Capturing Components

The Component Manager allows your component to capture another component. When a component is captured, the Component Manager removes the captured component from its list of available components. The `FindNextComponent` function does not return information about captured components. Also, other applications or clients cannot open or access captured components unless they have previously received a component identifier or component instance for the captured component. The routines described in this section allow your component to capture and uncapture other components.

Typically, your component captures another component when you want to override all or some of the features provided by a component or to provide new features. For example, a component called NewMath might capture a component called OldMath. Suppose the NewMath component provides a new function, `DoExponent`. Whenever NewMath gets an exponent request, it can handle the request itself. For all other requests, NewMath might call the OldMath component to perform the request.

After capturing a component, your component might choose to target a particular instance of the captured component. For information on targeting a component instance, see "Responding to the Target Request" beginning on page 6-25 and "Targeting a Component Instance" on page 6-77.

Use the `CaptureComponent` function to capture a component. Use the `UncaptureComponent` function to restore a previously captured component to the search list.

CaptureComponent

The `CaptureComponent` function allows your component to capture another component. In response to this function, the Component Manager removes the specified component from the search list of components. As a result, applications cannot retrieve information about the captured component or gain access to it. Current clients of the captured component are not affected by this function.

```
FUNCTION CaptureComponent (capturedComponent: Component;
                           capturingComponent: Component)
                           : Component;
```

`capturedComponent`
: The component identifier of the component to be captured. Your component can obtain this identifier from the `FindNextComponent` function or from the component registration routines.

`capturingComponent`
: The component identifier of your component. Note that you can use the component instance (appropriately coerced) that your component received in its open request in this parameter.

DESCRIPTION

The CaptureComponent function removes the specified component from the search list of components and returns a new component identifier. Your component can use this new identifier to refer to the captured component. For example, your component can open the captured component by providing this identifier to the OpenComponent function. Your component must provide this identifier to the UncaptureComponent function to specify the component to be restored to the search list.

If the component specified by the capturedComponent parameter is already captured, the CaptureComponent function returns a component identifier set to NIL.

SEE ALSO

See "Responding to the Target Request" on page 6-25 and "Targeting a Component Instance" on page 6-77 for information about target requests. For information related to the Component Manager's use of its list of available components, see page 6-42 for details on the FindNextComponent function and page 6-45 for details on the OpenDefaultComponent function. See "Registering Components" beginning on page 6-57 for details of the component registration routines.

UncaptureComponent

The UncaptureComponent function allows your component to uncapture a previously captured component.

```
FUNCTION UncaptureComponent (aComponent: Component): OSErr;
```

aComponent
: The component identifier of the component to be uncaptured. Your component obtains this identifier from the CaptureComponent function.

DESCRIPTION

The UncaptureComponent function restores the specified component to the search list of components. Applications can then access the component and retrieve information about the component using Component Manager routines.

RESULT CODES

noErr	0	No error
invalidComponentID	–3000	No component has this component identifier
componentNotCaptured	–3002	This component has not been captured

Targeting a Component Instance

Your component can target a component instance without capturing the component or your component can first capture the component and then target a specific instance of the component. For information on capturing components, see "Capturing Components" beginning on page 6-75. To target a component instance, use the `ComponentSetTarget` function.

ComponentSetTarget

You can use the `ComponentSetTarget` function to call a component's target request routine (that is, the routine that handles the `kComponentTargetSelect` request code). The target request informs a component that it has been targeted by another component.

You should not target a component instance if the component does not support the target request. Before calling this function, you should issue a can do request to the component instance you want to target to verify that the component supports the target request. If the component supports it, use the `ComponentSetTarget` function to send a target request to the component instance you wish to target. After receiving a target request, the targeted component instance should call the component instance that targeted it whenever the targeted component instance would normally call one of its defined functions.

```
FUNCTION ComponentSetTarget (ci: ComponentInstance;
                             target: ComponentInstance): LongInt;
```

`ci` The component instance to which to send a target request (the component that has been targeted).

`target` The component instance of the component issuing the target request.

DESCRIPTION

The `ComponentSetTarget` function returns a function result of `badComponentSelector` if the targeted component does not support the target request. Otherwise, the `ComponentSetTarget` function returns as its function result the value that the targeted component instance returned in response to the target request.

SEE ALSO

For details on how to handle the target request, see "Responding to the Target Request" on page 6-25.

Changing the Default Search Order

You can use the `SetDefaultComponent` function to change the order in which the list of registered components is searched.

SetDefaultComponent

The `SetDefaultComponent` function allows your component to change the search order for registered components. You specify a component that is to be placed at the front of the search chain, along with control information that governs the reordering operation. The order of the search chain influences which component the Component Manager selects in response to an application's use of the `OpenDefaultComponent` and `FindNextComponent` functions.

```
FUNCTION SetDefaultComponent (aComponent: Component;
                              flags: Integer): OSErr;
```

`aComponent`
A component identifier that specifies the component for this operation.

`flags` A value specifying the control information governing the operation. The value of this parameter controls which component description fields the Component Manager examines during the reorder operation. Set the appropriate flags to 1 to define the fields that are examined during the reorder operation. The following flags are defined:

`defaultComponentIdentical`
The Component Manager places the specified component in front of all other components that have the same component description.

`defaultComponentAnyFlags`
The Component Manager ignores the value of the `componentFlags` field during the reorder operation.

`defaultComponentAnyManufacturer`
The Component Manager ignores the value of the `componentManufacturer` field during the reorder operation.

`defaultComponentAnySubType`
The Component Manager ignores the value of the `componentSubType` field during the reorder operation.

DESCRIPTION

The `SetDefaultComponent` function changes the search order of registered components by moving the specified component to the front of the search chain, according to the value specified in the `flags` parameter.

SPECIAL CONSIDERATIONS

Note that the `SetDefaultComponent` function changes the search order for all applications. As a result, you should use this function carefully.

RESULT CODES

`noErr`	0	No error
`invalidComponentID`	–3000	No component has this component identifier

Application-Defined Routine

To provide a component, you define a component function and supply the appropriate registration information. You store your component function in a code resource and typically store your component's registration information as resources in a component file. For additional information on this process, see "Creating Components" beginning on page 6-13.

MyComponent

Here's how to declare a component function named `MyComponent`:

```
FUNCTION MyComponent (params: ComponentParameters;
                      storage: Handle): ComponentResult;
```

`params` A component parameters record. The `what` field of the component parameters record indicates the action your component should perform. The parameters that the client invoked your function with are contained in the `params` field of the component parameters record. Your component can use the `CallComponentFunction` or `CallComponentFunctionWithStorage` routine to extract the parameters from this record.

`storage` A handle to any memory that your component has associated with the connection. Typically, upon receiving an open request, your component allocates memory and uses the `SetComponentInstanceStorage` function to associate the allocated memory with the component connection.

DESCRIPTION

When your component receives a request, it should perform the action specified in the `what` field of the component parameters record. Your component should return a value of type `ComponentResult` (a long integer). If your component does not return error information as its function result, it should indicate errors using the `SetComponentInstanceError` procedure.

SEE ALSO

For information on the component parameters record, see page 6-54. For information on writing a component, see "Creating Components" beginning on page 6-13.

Resources

This section describes the resource you use to define your component. If you are developing a component, you should be familiar with the format and content of a component resource.

The Component Resource

A component resource (a resource of type `'thng'`) stores all of the information about a component in a single file. The component resource contains all the information needed to register a code resource as a component. Information in the component resource tells the Component Manager where to find the code for the component.

If you are developing an application that uses components, you do not need to know about component resources.

If you are developing a component or an application that registers components, you should be familiar with component resources. The Component Manager automatically registers any components that are stored in component files in the Extensions folder. The file type for component files must be set to `'thng'`. If you store your component in a component file in the Extensions folder, you do not need to create an application to register the component.

The Component Manager provides routines that register components. The `RegisterComponent` function registers components that are not stored in resource files. The `RegisterComponentResource` and `RegisterComponentResourceFile` functions register components that are stored as component resources in a component file. If you are developing an application that registers components, you should use the routine that is appropriate to the storage format of the component. For more information about how your application can register components, see "Registering Components" beginning on page 6-57.

This section describes the component resource, which must be provided by all components stored in a component file. Applications that register a component using the `RegisterComponent` function must also provide the same information as that contained in a component resource.

IMPORTANT

For compatibility with early versions of the Component Manager, a component resource must be locked. ▲

The `ComponentResource` data type defines the structure of a component resource. (You can also optionally append to the end of this structure the information defined by the `ComponentResourceExtension` data type, as shown in Figure 6-5 on page 6-85.)

```
ComponentResource =
   RECORD
      cd:                             {registration information}
                     ComponentDescription;
      component:     ResourceSpec;   {code resource}
      componentName: ResourceSpec;   {name string resource}
      componentInfo: ResourceSpec;   {info string resource}
      componentIcon: ResourceSpec;   {icon resource}
   END;
```

Field descriptions

`cd` A component description record that specifies the characteristics of the component. For a complete description of this record, see page 6-52.

`component` A resource specification record that specifies the type and ID of the component code resource. The `resType` field of the resource specification record may contain any value. The component's main entry point must be at offset 0 in the resource.

`componentName` A resource specification record that specifies the resource type and ID for the name of the component. This is a Pascal string. Typically, the component name is stored in a resource of type `'STR '`.

`componentInfo` A resource specification record that specifies the resource type and ID for the information string that describes the component. This is a Pascal string. Typically, the information string is stored in a resource of type `'STR '`. You might use the information stored in this resource in a Get Info dialog box.

`componentIcon` A resource specification record that specifies the resource type and ID for the icon for a component. Component icons are stored as 32-by-32 bit maps. Typically, the icon is stored in a resource of type `'ICON'`. Note that this icon is not used by the Finder; you supply an icon only so that other components or applications can display your component's icon in a dialog box if needed.

A resource specification record, defined by the data type `ResourceSpec`, describes the resource type and resource ID of the component's code, name, information string, or icon. The resources specified by the resource specification records must reside in the same resource file as the component resource itself.

```
ResourceSpec =
   RECORD
      resType:       OSType;        {resource type}
      resId:         Integer;       {resource ID}
   END;
```

You can optionally include in your component resource the information defined by the `ComponentResourceExtension` data type:

```
ComponentResourceExtension =
   RECORD
      componentVersion:       LongInt; {version of component}
      componentRegisterFlags: LongInt; {additional flags}
      componentIconFamily:    Integer; {resource ID of icon }
                                       { family}
   END;
```

Field descriptions

`componentVersion`
: The version number of the component. If you specify the `componentDoAutoVersion` flag in `componentRegisterFlags`, the Component Manager must obtain the version number of your component when your component is registered. Either you can provide a version number in your component's resource, or you can specify a value of 0 for its version number. If you specify 0, the Component Manager sends your component a version request to get the version number of your component.

`componentRegisterFlags`
: A set of flags containing additional registration information. You can use these constants as flags:

```
CONST
   componentDoAutoVersion            = 1;
   componentWantsUnregister          = 2;
   componentAutoVersionIncludeFlags = 4;
```

Specify the componentDoAutoVersion flag if you want the Component Manager to resolve conflicts between different versions of the same component. If you specify this flag, the Component Manager registers your component only if there is no later version available. If an older version is already registered, the Component Manager unregisters it. If a newer version of the same component is registered after yours, the Component Manager automatically unregisters your component. You can use this automatic version control feature to make sure that the most recent version of your component is registered, regardless of the number of versions that are installed.

Specify the componentWantsUnregister flag if you want your component to receive an unregister request when it is unregistered.

Specify the flag componentAutoVersionIncludeFlags if you want the Component Manager to include the componentFlags field of the component description record when it searches for identical components in the process of performing automatic version control for your component. If you do not specify this flag, the Component Manager searches only the componentType, componentSubType, and componentManufacturer fields.

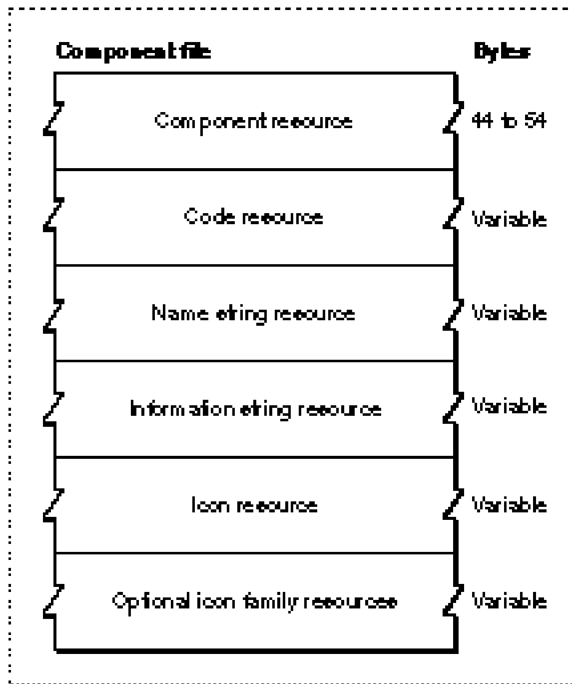
When the Component Manager performs automatic version control for your component, it searches for components with identical values in the componentType, componentSubType, and componentManufacturer fields (and optionally, in the componentFlags field). If it finds a matching component, it compares version numbers and registers the most recent version of the component. Note that the setting of the componentAutoVersionIncludeFlags flag affects automatic version control only and does not affect the search operations performed by FindNextComponent and CountComponents.

componentIconFamily

The resource ID of an icon family. You can provide an icon family in addition to the icon provided in the componentIcon field. Note that members of this icon family are not used by the Finder; you supply an icon family only so that other components or applications can display your component's icon in a dialog box if needed.

You store a component resource, along with other resources for the component, in the resource fork of a component file. Figure 6-4 shows the structure of a component file.

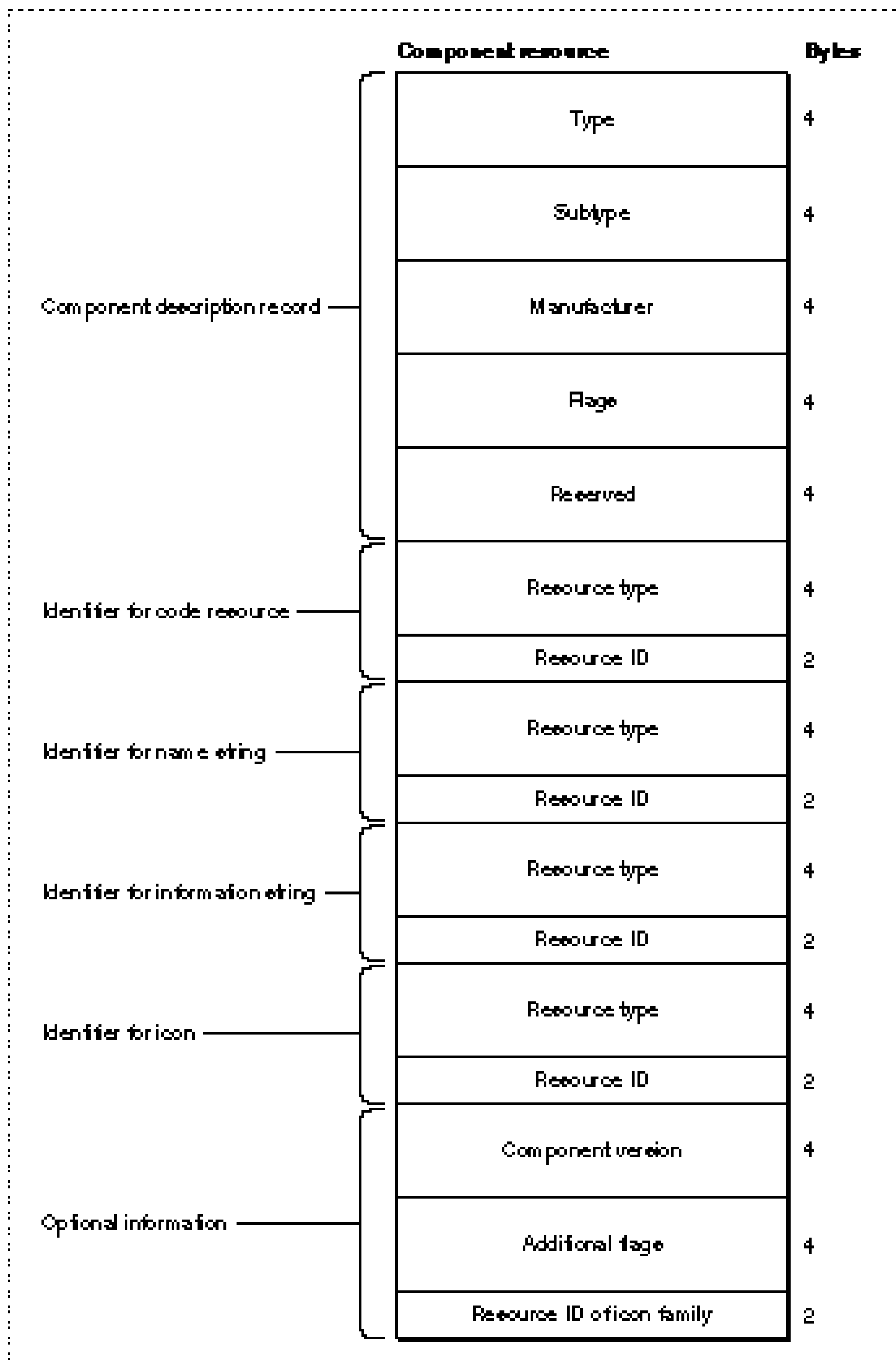
Figure 6-4 Format of a component file



You can also store other resources for your component in your component file. For example, you should include `'FREF'`, `'BNDL'`, and icon family resources so that the Finder can associate the icon identifying your component with your component file. When designing the icon for your component file, you should follow the same guidelines as those for system extension icons. See *Macintosh Human Interface Guidelines* for information on designing an icon. See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on the `'FREF'` and `'BNDL'` resources.

Figure 6-5 shows the structure of a component resource.

Figure 6-5 Structure of a compiled component ('thng') resource



Summary of the Component Manager

Pascal Summary

Constants

```
CONST
      gestaltComponentMgr              = 'cpnt';

      kComponentOpenSelect             = -1;    {open request}
      kComponentCloseSelect            = -2;    {close request}
      kComponentCanDoSelect            = -3;    {can do request}
      kComponentVersionSelect          = -4;    {version request}
      kComponentRegisterSelect         = -5;    {register request}
      kComponentTargetSelect           = -6;    {target request}
      kComponentUnregisterSelect       = -7;    {unregister request}

      {wildcard values for searches}
      kAnyComponentType                = 0;     {any type}
      kAnyComponentSubType             = 0;     {any subtype}
      kAnyComponentManufacturer        = 0;     {any manufacturer}
      kAnyComponentFlagsMask           = 0;     {any flags}

      {component description flag}
      cmpWantsRegisterMessage          = $80000000; {send register request}
      {flags for optional extension to component resource}
      componentDoAutoVersion           = 1;     {provide version control}
      componentWantsUnregister         = 2;     {send unregister request}
      componentAutoVersionIncludeFlags = 4;     {include flags in search}

      {flags for SetDefaultComponent function}
      defaultComponentIdentical        = 0;
      defaultComponentAnyFlags         = 1;
      defaultComponentAnyManufacturer  = 2;
      defaultComponentAnySubType       = 4;
      defaultComponentAnyFlagsAnyManufacturer
                                       = defaultComponentAnyFlags +
                                          defaultComponentAnyManufacturer;
```

```
      defaultComponentAnyFlagsAnyManufacturerAnySubType
                  = defaultComponentAnyFlags
                     + defaultComponentAnyManufacturer
                     + defaultComponentAnySubType;

{flags for the global parameter of RegisterComponentResourceFile function}
   registerCmpGlobal       = 1;  {other apps can communicate with component}
   registerCmpNoDuplicates = 2;  {don't register if duplicate component }
                                 { exists}
   registerCompAfter       = 4;  {component registered after all others of }
                                 { same type}
```

Data Types

```
TYPE
      ComponentDescription =
      RECORD
         componentType:                OSType;  {type}
         componentSubType:             OSType;  {subtype}
         componentManufacturer:        OSType;  {manufacturer}
         componentFlags:               LongInt; {control flags}
         componentFlagsMask:           LongInt; {mask for control flags }
                                                { (reserved when }
                                                { registering a component)}
      END;

      ResourceSpec =
      RECORD
         resType:                      OSType;  {resource type}
         resID:                        Integer; {resource ID}
      END;

      ComponentResourcePtr                   = ^ComponentResource;
      ComponentResourceHandle                = ^ComponentResourcePtr;
      ComponentResource =                          {component resource}
      RECORD
         cd:               ComponentDescription;   {registration information}
         component:        ResourceSpec;           {code resource}
         componentName:    ResourceSpec;           {name string resource}
         componentInfo:    ResourceSpec;           {info string resource}
         componentIcon:    ResourceSpec;           {icon resource}
      END;
```

```
ComponentResourceExtension =           {optional extension to resource}
RECORD
   componentVersion:       LongInt; {version of component}
   componentRegisterFlags: LongInt; {additional flags}
   componentIconFamily:    Integer; {resource ID of icon }
                                    { family}
END;
{component parameters record}
ComponentParameters =
   PACKED RECORD
      flags:      Char;                   {reserved}
      paramSize:  Char;                   {size in bytes of actual }
                                          { parameters passed to }
                                          { this routine}
      what:       Integer;                {request code- }
                                          { negative for requests }
                                          { defined by Component Mgr}
      params:     ARRAY[0..0] OF LongInt; {actual parameters for }
                                          { the indicated routine}
   END;

{component identifier}
Component         = ^ComponentRecord;
ComponentRecord   =
RECORD
   data:       ARRAY[0..0] OF LongInt;
END;

{component instance}
ComponentInstance = ^ComponentInstanceRecord;
ComponentInstanceRecord =
RECORD
   data:    ARRAY[0..0] OF LongInt;
END;

ComponentResult   = LongInt;
ComponentRoutine  = ProcPtr;
ComponentFunction = ProcPtr;
```

Routines for Applications

Finding Components

```
FUNCTION FindNextComponent   (aComponent: Component;
                              looking: ComponentDescription): Component;
FUNCTION CountComponents     (looking: ComponentDescription): LongInt;
FUNCTION GetComponentListModSeed: LongInt;
```

Opening and Closing Components

```
FUNCTION OpenDefaultComponent
                             (componentType: OSType;
                              componentSubType: OSType): ComponentInstance;
FUNCTION OpenComponent       (aComponent: Component): ComponentInstance;
FUNCTION CloseComponent      (aComponentInstance: ComponentInstance): OSErr;
```

Getting Information About Components

```
FUNCTION GetComponentInfo    (aComponent: Component;
                              VAR cd: ComponentDescription;
                              componentName: Handle; componentInfo: Handle;
                              componentIcon: Handle): OSErr;
FUNCTION GetComponentIconSuite
                             (aComponent: Component;
                              VAR iconSuite: Handle): OSErr;
FUNCTION GetComponentVersion
                             (ci: ComponentInstance): LongInt;
FUNCTION ComponentFunctionImplemented
                             (ci: ComponentInstance; ftnNumber: Integer)
                              : LongInt;
```

Retrieving Component Errors

```
FUNCTION GetComponentInstanceError
                             (aComponentInstance: ComponentInstance): OSErr;
```

Routines for Components

Registering Components

```
FUNCTION RegisterComponent   (cd: ComponentDescription;
                              componentEntryPoint: ComponentRoutine;
                              global: Integer; componentName: Handle;
                              componentInfo: Handle;
                              componentIcon: Handle): Component;
FUNCTION RegisterComponentResource
                             (cr: ComponentResourceHandle;
                              global: Integer): Component;
FUNCTION RegisterComponentResourceFile
                             (resRefNum: integer; global: integer); LongInt;
FUNCTION UnregisterComponent
                             (aComponent: Component): OSErr;
```

Dispatching to Component Routines

```
FUNCTION CallComponentFunction
                             (params: ComponentParameters;
                              func: ComponentFunction): LongInt;
FUNCTION CallComponentFunctionWithStorage
                             (storage: Handle;
                              params: ComponentParameters;
                              func: ComponentFunction): LongInt;
```

Managing Component Connections

```
PROCEDURE SetComponentInstanceStorage
                             (aComponentInstance: ComponentInstance;
                              theStorage: Handle);
FUNCTION GetComponentInstanceStorage
                             (aComponentInstance: ComponentInstance): Handle;
FUNCTION CountComponentInstances
                             (aComponent: Component): LongInt;
PROCEDURE SetComponentInstanceA5
                             (aComponentInstance: ComponentInstance;
                              theA5: LongInt);
FUNCTION GetComponentInstanceA5
                             (aComponentInstance: ComponentInstance)
                              : LongInt;
```

Setting Component Errors

```
PROCEDURE SetComponentInstanceError
                            (aComponentInstance: ComponentInstance;
                             theError: OSErr);
```

Working With Component Reference Constants

```
PROCEDURE SetComponentRefcon
                            (aComponent: Component; theRefcon: LongInt);
FUNCTION GetComponentRefcon
                            (aComponent: Component): LongInt;
```

Accessing a Component's Resource File

```
FUNCTION OpenComponentResFile
                            (aComponent: Component): Integer;
FUNCTION CloseComponentResFile
                            (refnum: Integer): OSErr;
```

Calling Other Components

```
FUNCTION DelegateComponentCall
                            (originalParams: ComponentParameters;
                             ci: ComponentInstance): LongInt;
```

Capturing Components

```
FUNCTION CaptureComponent   (capturedComponent: Component;
                             capturingComponent: Component): Component;
FUNCTION UncaptureComponent
                            (aComponent: Component): OSErr;
```

Targeting a Component Instance

```
FUNCTION ComponentSetTarget (ci: ComponentInstance;
                             target: ComponentInstance): LongInt;
```

Changing the Default Search Order

```
FUNCTION SetDefaultComponent
                            (aComponent: Component; flags: Integer): OSErr;
```

Application-Defined Routine

```
FUNCTION MyComponent        (params: ComponentParameters;
                             storage: Handle): ComponentResult;
```

C Summary

Constants

```
   #define gestaltComponentMgr 'cpnt'        /*Gestalt selector*/

   /*required component routines*/
   #define kComponentOpenSelect          -1   /*open request*/
   #define kComponentCloseSelect         -2   /*close request*/
   #define kComponentCanDoSelect         -3   /*can do request*/
   #define kComponentVersionSelect       -4   /*version request*/
   #define kComponentRegisterSelect      -5   /*register request*/
   #define kComponentTargetSelect        -6   /*target request*/
   #define kComponentUnregisterSelect    -7   /*unregister request*/

   /*wildcard values for searches*/
   #define kAnyComponentType             0    /*any type*/
   #define kAnyComponentSubType          0    /*any subtype*/
   #define kAnyComponentManufacturer     0    /*any manufacturer*/
   #define kAnyComponentFlagsMask        0    /*any flags*/

/*component description flags*/
enum {
   cmpWantsRegisterMessage = 1L<<31          /*send register request*/
};
/*flags for optional extension to component resource*/
enum {
   componentDoAutoVersion            = 1,     /*provide version control*/
   componentWantsUnregister          = 2,     /*send unregister request*/
   componentAutoVersionIncludeFlags = 4       /*include flags in search*/
};
enum {   /*flags for SetDefaultComponent function*/
   defaultComponentIdentical         = 0,
   defaultComponentAnyFlags          = 1,
   defaultComponentAnyManufacturer  = 2,
```

```
   defaultComponentAnySubType       = 4,
};
   #define defaultComponentAnyFlagsAnyManufacturer
                  (defaultComponentAnyFlags+defaultComponentAnyManufacturer)
   #define defaultComponentAnyFlagsAnyManufacturerAnySubType
                  (defaultComponentAnyFlags+defaultComponentAnyManufacturer
                  +defaultComponentAnySubType)

enum {
/*flags for the global parameter of RegisterComponentResourceFile function*/
   registerCmpGlobal        = 1, /*other apps can communicate with */
                                 /* component*/
   registerCmpNoDuplicates  = 2, /*duplicate component exists*/
   registerCompAfter        = 4  /*component registered after all others */
                                 /* of same type*/
};
```

Data Structures

```
struct ComponentDescription {
   OSType         componentType;          /*type*/
   OSType         componentSubType;       /*subtype*/
   OSType         componentManufacturer;  /*manufacturer*/
   unsigned long  componentFlags;         /*control flags*/
   unsigned long  componentFlagsMask;     /*mask for control flags */
                                          /* (reserved when registering */
                                          /* a component)*/
};
typedef struct ComponentDescription ComponentDescription;

struct ResourceSpec {
   OSType         ResType;                /*resource type*/
   short          ResID;                  /*resource ID*/
};
typedef struct ResourceSpec ResourceSpec;
```

```
struct ComponentResource {
   ComponentDescription cd;             /*registration information*/
   ResourceSpec         component;      /*code resource*/
   ResourceSpec         componentName;  /*name string resource*/
   ResourceSpec         componentInfo;  /*info string resource*/
   ResourceSpec         componentIcon;  /*icon resource*/
};
typedef struct ComponentResource ComponentResource;
typedef ComponentResource *ComponentResourcePtr, **ComponentResourceHandle;

/*optional extension to component resource*/
struct ComponentResourceExtension {
   long           componentVersion;       /*version number*/
   long           componentRegisterFlags; /*additional flags*/
   short          componentIconFamily;    /*resource ID of icon family*/
};
typedef struct ComponentResourceExtension ComponentResourceExtension;
/*structure received by component*/
struct ComponentParameters {
   unsigned char flags;       /*reserved*/
   unsigned char paramSize;   /*size in bytes of actual parameters passed */
                              /* to this routine*/
   short        what;         /*request code, negative for requests */
                              /* defined by Component Mgr*/
   long         params[1];    /*actual parameters for the indicated */
                              /* routine*/
};
typedef struct ComponentParameters ComponentParameters;

/*component identifier*/
typedef struct privateComponentRecord *Component;
/*component instance*/
typedef struct privateComponentInstanceRecord *ComponentInstance;

typedef long ComponentResult;

typedef pascal ComponentResult (*ComponentRoutine)
      (ComponentParameters *cp, Handle componentStorage);
typedef pascal ComponentResult (*ComponentFunction)();

#define ComponentCallNow(callNumber, paramSize) \
         {0x2F3C, paramSize, callNumber, 0x7000, 0xA82A}
```

Routines for Applications

Finding Components

```
pascal Component FindNextComponent
                             (Component aComponent,
                              ComponentDescription *looking);
pascal long CountComponents
                             (ComponentDescription *looking);
pascal long GetComponentListModSeed
                             (void);
```

Opening and Closing Components

```
pascal ComponentInstance OpenDefaultComponent
                             (OSType componentType,
                              OSType componentSubType);
pascal ComponentInstance OpenComponent
                             (Component aComponent);
pascal OSErr CloseComponent
                             (ComponentInstance aComponentInstance);
```

Getting Information About Components

```
pascal OSErr GetComponentInfo
                             (Component aComponent,
                              ComponentDescription *cd,
                              Handle componentName, Handle componentInfo,
                              Handle componentIcon);
pascal OSErr GetComponentIconSuite
                             (Component aComponent,
                              Handle *iconSuite);
pascal long GetComponentVersion
                             (ComponentInstance ci);
pascal long ComponentFunctionImplemented
                             (ComponentInstance ci, short ftnNumber);
```

Retrieving Component Errors

```
pascal OSErr GetComponentInstanceError
                             (ComponentInstance aComponentInstance);
```

Routines for Components

Registering Components

```
pascal Component RegisterComponent
                              (ComponentDescription *cd,
                               ComponentRoutine componentEntryPoint,
                               short global, Handle componentName,
                               Handle componentInfo, Handle componentIcon);
pascal Component RegisterComponentResource
                              (ComponentResourceHandle cr, short global);
pascal long RegisterComponentResourceFile
                              (short resRefNum, short global);
pascal OSErr UnregisterComponent
                              (Component aComponent);
```

Dispatching to Component Routines

```
pascal long CallComponentFunction
                              (ComponentParameters *params,
                               ComponentFunction func);
pascal long CallComponentFunctionWithStorage
                              (Handle storage, ComponentParameters *params,
                               ComponentFunction func);
```

Managing Component Connections

```
pascal void SetComponentInstanceStorage
                              (ComponentInstance aComponentInstance,
                               Handle theStorage);
pascal Handle GetComponentInstanceStorage
                              (ComponentInstance aComponentInstance);
pascal long CountComponentInstances
                              (Component aComponent);
pascal void SetComponentInstanceA5
                              (ComponentInstance aComponentInstance,
                               long theA5);
pascal long GetComponentInstanceA5
                              (ComponentInstance aComponentInstance);
```

Setting Component Errors

```
pascal void SetComponentInstanceError
                              (ComponentInstance aComponentInstance,
                               OSErr theError);
```

Working With Component Reference Constants

```
pascal void SetComponentRefcon
                            (Component aComponent, long theRefcon);
pascal long GetComponentRefcon
                            (Component aComponent);
```

Accessing a Component's Resource File

```
pascal short OpenComponentResFile
                            (Component aComponent);
pascal OSErr CloseComponentResFile
                            (short refnum);
```

Calling Other Components

```
pascal long DelegateComponentCall
                            (ComponentParameters *originalParams,
                             ComponentInstance ci);
```

Capturing Components

```
pascal Component CaptureComponent
                            (Component capturedComponent,
                             Component capturingComponent);
pascal OSErr UncaptureComponent
                            (Component aComponent);
```

Targeting a Component Instance

```
pascal long ComponentSetTarget
                            (ComponentInstance ci,
                             ComponentInstance target);
```

Changing the Default Search Order

```
pascal OSErr SetDefaultComponent
                            (Component aComponent, short flags);
```

Application-Defined Routine

```
pascal ComponentResult MyComponent
                            (ComponentParameters* params,
                             Handle storage);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

_ComponentDispatch

Selector	Routine
$7001	RegisterComponent
$7002	UnregisterComponent
$7003	CountComponents
$7004	FindNextComponent
$7005	GetComponentInfo
$7006	GetComponentListModSeed
$7007	OpenComponent
$7008	CloseComponent
$700A	GetComponentInstanceError
$700B	SetComponentInstanceError
$700C	GetComponentInstanceStorage
$700D	SetComponentInstanceStorage
$700E	GetComponentInstanceA5
$700F	SetComponentInstanceA5
$7010	GetComponentRefcon
$7011	SetComponentRefcon
$7012	RegisterComponentResource
$7013	CountComponentInstances
$7014	RegisterComponentResourceFile
$7015	OpenComponentResFile
$7018	CloseComponentResFile
$701C	CaptureComponent
$701D	UncaptureComponent
$701E	SetDefaultComponent
$7021	OpenDefaultComponent
$7024	DelegateComponentCall
$70FF	CallComponentFunction
$70FF	CallComponentFunctionWithStorage

Result Codes

noErr	0	No error
resFNotFound	–193	Resource file not found
invalidComponentID	–3000	No component has this component identifier
validInstancesExist	–3001	This component has open connections
componentNotCaptured	–3002	This component has not been captured
badComponentInstance	$800008001	Invalid component passed to Component Manager
badComponentSelector	$800008002	Component does not support the specified request code