This chapter describes how to use the Resource Manager to read and write resources. You typically use resources to store the descriptions for user interface elements such as menus, windows, controls, dialog boxes, and icons. In addition, your application can store variable settings, such as the location of a window at the time the user closes the window, in a resource. When the user opens the document again, your application can read the information in the resource and restore the window to its previous location.

This chapter begins with an introduction to basic concepts you should understand before you begin to use Resource Manager routines. The rest of the chapter describes how to

- create resources
- get a handle to a resource
- release and detach resources
- create and open a resource fork
- set the current resource file
- read and manipulate resources
- write resources
- read and write partial resources

To use this chapter, you should be familiar with basic memory management on Macintosh computers and the Memory Manager. See the chapter "Introduction to Memory Management" in *Inside Macintosh: Memory* for details. You should also be familiar with the File Manager and the Standard File Package. See *Inside Macintosh: Files* for this information.

For information on how to create resources using a high-level resource editor like the ResEdit application or a resource compiler like Rez, see *ResEdit Reference* and *Macintosh Programmer's Workshop Reference*. (Rez is provided with Apple's Macintosh Programmer's Workshop [MPW]; both MPW and ResEdit are available through APDA.)

To get information on the format of an individual resource type, see the documentation for the manager that interprets that resource. For example, to get the format of a `'MENU'` resource, refer to the chapter "Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

## Introduction to Resources

A **resource** is data of any kind stored in a defined format in a file's resource fork. The Resource Manager keeps track of resources in memory and allows your application to read or write resources.

Resources are a basic element of every Macintosh application. Resources typically include data that describes menus, windows, controls, dialog boxes, sounds, fonts, and icons. Because such resources are separate from the application's code, you can easily create and manage resources for menu titles, dialog boxes, and other parts of your application without recompiling. Resources also simplify the process of translating interface elements containing text into other languages.

Applications and system software interpret the data for a resource according to its resource type. You usually create resources using a resource compiler or resource editor. This book shows resources in Rez format (Rez is a resource compiler provided with MPW). You can also use other resource tools, such as ResEdit, to create the resources for your application.

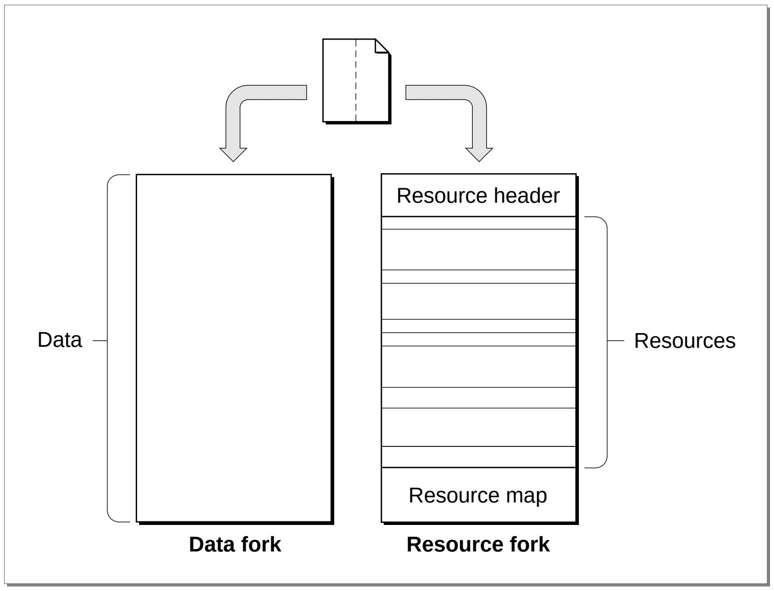
*Inside Macintosh: Macintosh Toolbox Essentials* describes how other managers, such as the Menu Manager, Window Manager, Dialog Manager, and Control Manager, use the Resource Manager to read resources for you. For example, you can use the Menu Manager, Window Manager, Dialog Manager, and Control Manager to read descriptions of your application's menus, windows, dialog boxes, and controls from resources. These managers all interpret a resource's data appropriately once it is read into memory. Although you'll typically use these managers to read resources for you, you can also use the Resource Manager directly to read and write resources.

## The Data Fork and the Resource Fork

In Macintosh system software, a **file** is a named, ordered sequence of bytes stored on a volume and divided into two forks, the data fork and the resource fork. The **data fork** usually contains data created by the user; the application creating the file can store and interpret the data in the data fork in whatever manner is appropriate. The **resource fork** of a file consists of a resource header, the resources themselves, and a resource map.

Figure 1-1 shows the data fork and resource fork of a file.

**Figure 1-1** The data fork and resource fork of a file



The resource header includes offsets to the beginning of the resource data and to the resource map. The resource map includes information about the resources in the resource fork and offsets to the location of each resource.

A Macintosh file always contains both a resource fork and a data fork, although one or both of those forks can be empty. The data fork of a document file typically contains data created by the user, and the resource fork contains any document-specific resources, such as preference settings and the document's last window position. The resource fork of an application file (that is, any file with the file type `'APPL'`) typically includes resources that describe the application's menus, windows, controls, dialog boxes, and icons, as well as the application's `'CODE'` resources. The resource fork of a file is also called a **resource file,** because in some respects you can treat it as if it were a separate file.

**IMPORTANT**

You should store all language-dependent data of your application, such as text used in help balloons and dialog boxes, as resources. If you do this, you can begin to localize your application by editing your application's resources without recompiling the application code. ▲

When your application writes data to a file, it writes to either the file's resource fork or its data fork. Typically, you use File Manager routines to read from and write to a file's data fork and Resource Manager routines to read from and write to a file's resource fork.

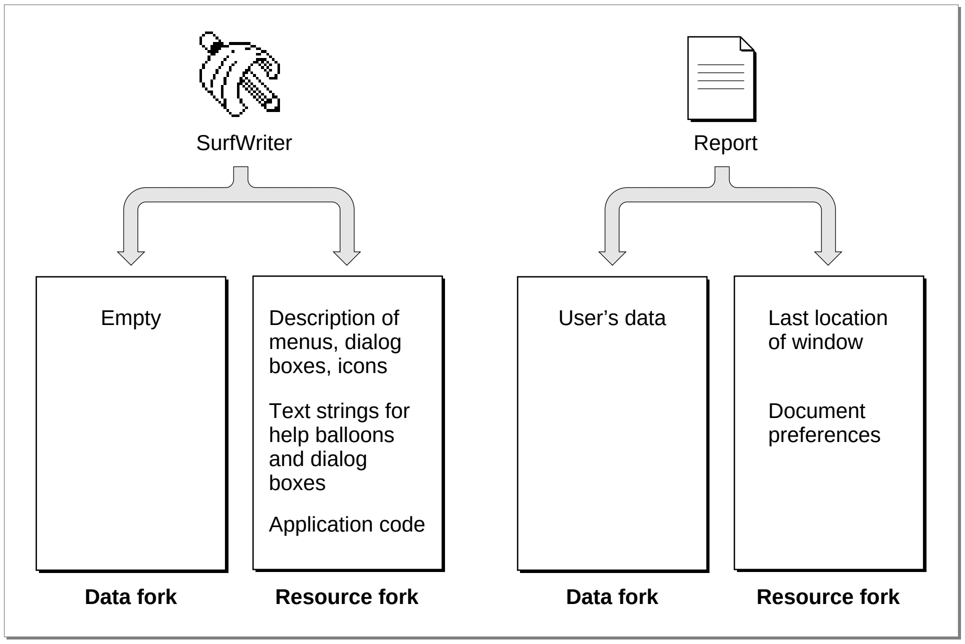
Whether you store data in the data fork or the resource fork of a document file depends largely on whether you can structure that data in a useful manner as a resource. For example, it's often convenient to store document-specific settings, such as the document's previous window size and location, as a resource in the document's resource fork. Data that the user is likely to edit is usually stored in the data fork of a document.

▲ **WARNING**

Don't use the resource fork of a file for data that is not in resource format. The Resource Manager assumes that any information in a resource fork can be interpreted according to the standard resource format described in this chapter. ▲

Figure 1-2 illustrates the typical contents of the data forks and resource forks of an application file and a document file.

**Figure 1-2** An application's and a document's data fork and resource fork



A resource fork can contain at most 2727 resources. The Resource Manager uses a linear search when searching a resource fork's resource types and resource IDs. In general, you should not create more than 500 resources of the same type in any one resource fork.

## Resource Types and Resource IDs

You typically use resources to store structured data, such as icons and sounds, and descriptions of menus, controls, dialog boxes, and windows. When you create a resource, you assign it a resource type and resource ID. A **resource type** is a sequence of four characters that uniquely identifies a specific type of resource, and a **resource ID** identifies a specific resource of a given type by number. (You can also use a resource name instead of a resource ID to identify a resource of a given type. However, a resource ID is preferred because it's generally more convenient to generate unique numbers than unique names.)

For example, to create a description of a menu in a resource, you create a resource of type `'MENU'` and give it a resource ID or resource name that differs from any other `'MENU'` resources that you have defined. In general, resource numbers 128 through 32767 are available for your use, although the numbers you can use for some types of resources are more restricted. (See "Resource IDs" on page 1-46 for more information about restrictions on the resource IDs used with specific resource types.)

System software defines a number of standard resource types. Here are some examples:

| Resource type | Description |
|---|---|
| 'ALRT' | Alert box |
| 'CNTL' | Control |
| 'CODE' | Application code segment |
| 'DITL' | Item list in a dialog box or alert box |
| 'DLOG' | Dialog box |
| 'ICN#' | Large (32-by-32 pixel) black-and-white icon, with mask |
| 'ICON' | Large (32-by-32 pixel) black-and-white icon, without mask |
| 'MBAR' | Menu bar |
| 'MENU' | Menu |
| 'NFNT' | Bitmapped font |
| 'STR ' | String |
| 'STR#' | String list |
| 'WIND' | Window |
| 'movv' | QuickTime movie |
| 'snd ' | Sound |

You can use these resource types to define their corresponding elements (for example, use a 'WIND' resource to define a window). You can also create your own resource types if your application needs resources other than the standard resource types defined by the system software. See Table 1-2 on page 1-43 for a complete list of standard resource types.

The Resource Manager does not interpret the format of an individual resource type. When you request a resource of a particular type with a given resource ID, the Resource Manager looks for the specified resource and, if it finds it, reads the resource into memory and returns a handle to it.

Your application or other system software routines can use the Resource Manager to read resources into memory. For example, when you use the Window Manager to read a description of a window from a 'WIND' resource, the Window Manager uses the Resource Manager to read the resource into memory. Once the resource is in memory, the Window Manager interprets the resource's data and creates a window with the characteristics described by the resource.

System software stores certain resources for its own use in the System file's resource fork. Although many of these resources are used only by the system software, your application can use some of them if necessary. For example, the standard images for the I-beam and wristwatch cursors are stored as resources of type 'CURS' in the System file. Your application can use these resources to change the appearance of the cursor.

## The Resource Map

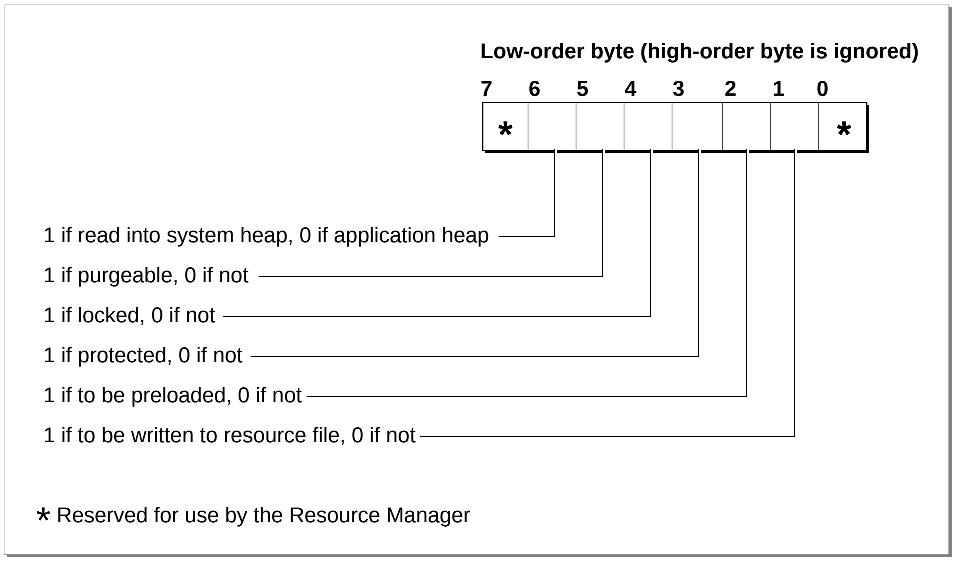
The **resource map** in the resource fork of a file contains entries for each resource in the resource fork. Each entry lists the resource's resource type, resource ID, name, attributes, and location. When the Resource Manager opens the resource fork of a file, it reads the resource map into memory. The resource map remains in memory until the file is closed.

The entries in the resource map on disk give the locations of resources as offsets to their locations in the resource fork. The entries in the resource map in memory specify the location of resources using handles—a handle whose value is `NIL`, if the resource is not currently in memory, or a handle to the resource's location in memory.

**Resource attributes** are flags that tell the Resource Manager how to handle the resource. For example, resource attributes specify whether the resource should be read into memory immediately when the Resource Manager opens the resource fork or read into memory only when needed; whether the resource should be read into the application or system heap; and whether the resource is purgeable.

The resource attributes for a resource are described by bits in the low-order byte of an integer value. Figure 1-3 shows which bits correspond to each resource attribute.

**Figure 1-3** Resource attributes


Low-order byte (high-order byte is ignored)
7 6 5 4 3 2 1 0
*
*
1 if read into system heap, 0 if application heap
1 if purgeable, 0 if not
1 if locked, 0 if not
1 if protected, 0 if not
1 if to be preloaded, 0 if not
1 if to be written to resource file, 0 if not
* Reserved for use by the Resource Manager

When it first opens a resource fork, the Resource Manager examines the resource attributes for each resource listed in the resource map. If the preloaded attribute of the resource is set, the Resource Manager reads the resource into memory and specifies its location by setting the resource's resource map entry in memory to contain a handle to the resource data. If the preloaded attribute of the resource is not set, the Resource Manager does not read the resource into memory; instead, it specifies the resource's location in the resource map entry in memory with a handle whose value is `NIL`.

When searching for a resource, the Resource Manager always looks in the resource map in memory, not the resource map of the resource fork on disk. If the resource map in memory specifies a handle for a particular resource, the Resource Manager uses the resource in memory; if the resource map in memory specifies a handle whose value is `NIL`, the Resource Manager reads the resource from the resource fork on disk into memory.

You can set the system heap attribute of a resource if you want to read a resource into the system heap. In most cases you should not set this attribute. If you do not set the system heap attribute, the Resource Manager reads the resource into relocatable blocks of your application's heap.

The purgeable attribute specifies whether the Resource Manager can purge a resource from memory to make room in memory for other data. If you specify that a resource is purgeable, you need to use the Resource Manager to make sure the resource is still in memory before referring to it through its resource handle.

Some resources must not be purgeable. For example, the Menu Manager expects menu resources to remain in memory, so you should not set the purgeable attribute of a menu resource. Other resources, such as windows, controls, and dialog boxes, do not have to remain in memory once the corresponding user interface element has been created. You should set the purgeable attribute for these kinds of resources.

You can set the locked attribute of a resource if you do not want the resource to be relocatable or purgeable. The locked attribute overrides the purgeable attribute; when the locked attribute is set, the resource is not purgeable, even if the purgeable attribute is set.

**Note**
If both the preloaded attribute and the locked attribute are set, the Resource Manager loads the resource as low in the heap as possible. ◆

You can set the protected attribute of a resource to ensure that your application doesn't accidentally change the resource ID or name of the resource, modify its contents, or remove the resource from its resource fork. In most cases you do not need to set this attribute. If you do set the protected attribute of a resource, you can still use a Resource Manager routine to change the protected attribute or to set other attributes of the resource.

The changed attribute applies only while the resource map is in memory. You should specify a value of 0 for the bit representing the changed attribute of a resource stored on disk. The Resource Manager sets the changed attribute of a resource's entry in the resource map in memory whenever your application changes a resource using the `ChangedResource` procedure, changes a resource map entry using the `SetResAttrs` or `SetResInfo` procedure, or adds a resource using the `AddResource` procedure.

## Search Path for Resources

When your application uses a Resource Manager routine to read or perform an operation on a resource, the Resource Manager follows a defined search path to find the resource. The file whose resource fork the Resource Manager searches first is referred to as the **current resource file.** Whenever your application opens a resource fork of a file, that file becomes the current resource file. Thus, the current resource file usually corresponds to the file whose resource fork was opened most recently. However, your application can change the current resource file if needed by using the `UseResFile` procedure.

Most of the Resource Manager routines assume that the current resource file is the file on whose resource fork they should operate or, in the case of a search, the resource fork in which to begin the search. If the Resource Manager can't find the resource in the current resource file, it continues searching until it either finds the resource or has searched all files in the search path.

On startup, system software calls the `InitResources` function to initialize the Resource Manager. The Resource Manager creates a special heap zone within the system heap and builds a resource map that points to ROM-resident resources. It opens the resource fork of the System file and reads its resource map into memory.

When a user opens your application, system software opens your application's resource fork. When your application opens a file, your application typically opens both the file's data fork and the file's resource fork. When the Resource Manager searches for a resource, it normally looks first in the resource map in memory of the last resource fork that your application opened. So, if your application has a single file open, the Resource Manager looks first in the resource map for that file's resource fork. If the Resource Manager doesn't find the resource there, it continues to search the resource maps of each resource fork open to your application in reverse order of opening (that is, the most recently opened is searched first). After looking in the resource maps of the resource files your application has opened, the Resource Manager searches your application's resource map. If it doesn't find the resource there, it searches the System file's resource map.

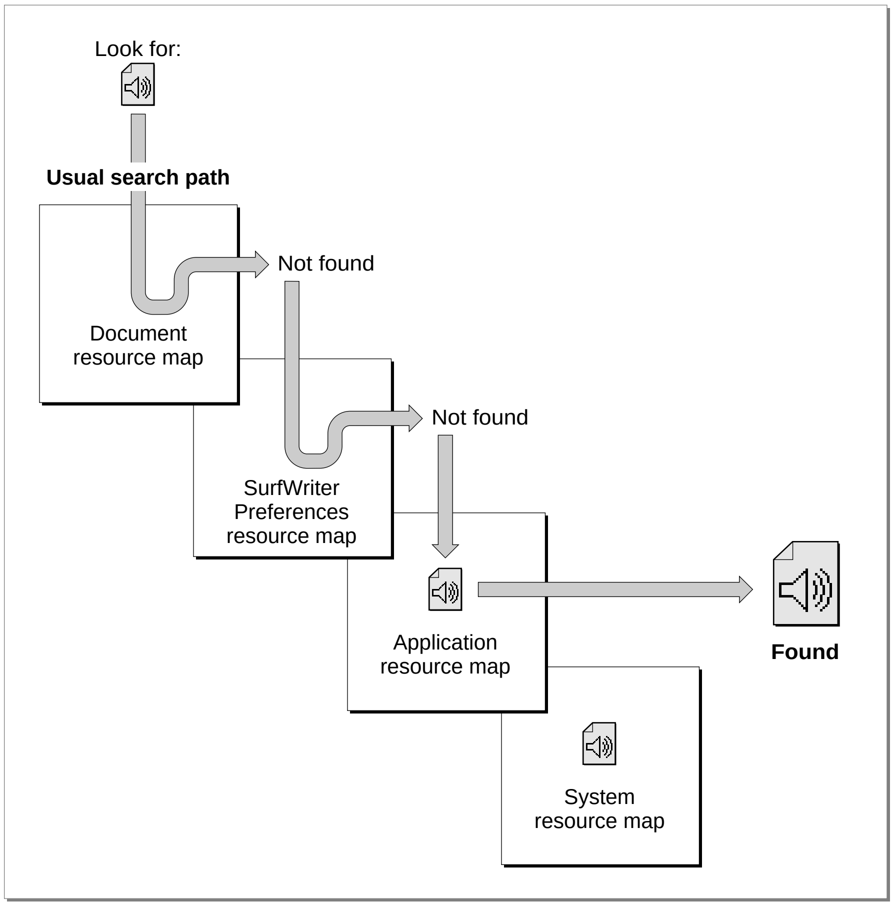
This default search order allows your application to use resources defined in the System file, to override resources defined in the System file, to share a single resource among several files by storing it in your application's resource fork, and to override application-defined resources with document-specific resources.

When the Resource Manager opens a resource fork, the File Manager assigns that resource fork a **file reference number,** which is a unique number identifying an access path to the resource fork. Your application needs to keep track of the file reference number of its own resource fork, so that it can refer specifically to that resource fork when necessary. Your application may also need to keep track of the file reference numbers for other resource forks that it opens.

For example, the SurfWriter application stores in its own resource fork the first few bars of Beethoven's Fifth Symphony as a resource of type `'snd '`. The SurfWriter application plays this sound whenever the user writes more than one page of text per hour. The user can change this sound for all documents created by SurfWriter by using SurfWriter's Preferences command to specify or record a new sound.

SurfWriter also allows the user to associate a sound with a specific document by using SurfWriter's Set Reward Sound command to specify or record a new sound. When SurfWriter wants to play the sound, it uses the Resource Manager to read the resource of type `'snd '` with the resource ID `kProductiveWriter`. Figure 1-4 shows the search path the Resource Manager takes to find this sound resource.

**Figure 1-4** A typical search order for a specific resource



System software opens SurfWriter's resource fork when the user opens the SurfWriter application. On startup, SurfWriter opens its preferences file (SurfWriter Preferences). When the user opens a SurfWriter document, SurfWriter opens the document's data fork and resource fork. When SurfWriter attempts to read an `'snd '` resource, the Resource Manager looks first in the resource map in memory of the current resource file (in the example illustrated in Figure 1-4, the SurfWriter document) for the requested resource. If the Resource Manager doesn't find the resource, it searches the resource map of the next most recently opened file (in this example, SurfWriter Preferences). It continues searching the resource forks in memory of any resource forks open to the SurfWriter application until it either finds the resource or has searched the last resource map in its search path. Typically the last resource map searched by the Resource Manager is the resource map of the System file. This allows your application to use resources in the System file as a default.

Table 1-1 summarizes the typical locations of resources used by an application.

**Table 1-1** Typical locations of resources

| Resource fork | Resources contained in resource fork |
|---|---|
| Resource fork of System file | Sounds, icons, cursors, and other elements available for use by all applications, and code resources that manage user interface elements such as menus, controls, and windows |
| Resource fork of application | Static data (such as text used in dialog boxes or help balloons) and descriptions of menus, windows, controls, icons, and other elements |
| Resource fork of application's preferences file | Data that encodes the user's global preferences for the application |
| Resource fork of document | Data that defines characteristics specific only to this document, such as its last size and location |

Although you can take advantage of the Resource Manager's search order to find a particular resource, in general your application should set the current resource file to the file whose resource fork contains the desired resource before reading and writing resource data. In addition, you can restrict the Resource Manager search path by using Resource Manager routines that look only in the current resource file's resource map when searching for a specific resource.

## About the Resource Manager

The Resource Manager provides routines that allow your application (and system software) to create, delete, open, read, modify, and write resources; get information about them; and alter the Resource Manager's search path.

Most Macintosh applications commonly read data from resources either indirectly, by calling other system software routines (such as Menu Manager routines) that in turn call the Resource Manager, or directly, by calling Resource Manager routines. At any time during your application's execution, at least two resource forks from which it can read information are likely to be open: the System file's resource fork and your application's resource fork.

As previously described, system software opens the System file's resource fork at startup and your application's resource fork at application launch. Your application is likely to open the resource forks of several other files at various times while it is running. For example, if your application saves the last position and size of a window (as determined by the user), you can use Resource Manager routines to write this information to an application-defined resource in the document file's resource fork. The next time the user opens the document, your application can use the Resource Manager to read the information saved in this resource and position the document accordingly.

You can store the user's general preferences, such as the default font or paper size, in your application's preferences file. You store a preferences file in the Preferences folder of the System Folder. The name of an application's preferences file typically consists of the name of the application followed by the word "Preferences." If your application can be shared by multiple users, you can use the Resource Manager to create a separate preferences file for each user.

## Using the Resource Manager

You use the Resource Manager to perform operations on resources. To determine whether certain features of the Resource Manager are available (support for `FSSpec` records and partial resources), use the `Gestalt` function.

Two commonly used Resource Manager routines use a file system specification (`FSSpec`) record: the `FSpCreateResFile` procedure and the `FSpOpenResFile` function. These routines are available only in System 7 or later. Call the `Gestalt` function with the `gestaltFSAttr` selector to determine whether the Resource Manager routines that use `FSSpec` records exist. If the bit indicated by the constant `gestaltHasFSSpecCalls` is set, then the routines are available.

```
CONST
   gestaltFSAttr          = 'fs  ';   {Gestalt selector for }
                                      { File Mgr attributes}
   gestaltHasFSSpecCalls  = 1;        {check this bit in the }
                                      { response parameter}
```

In addition, the Resource Manager routines for reading and writing partial resources are available only in System 7 or later versions of system software. Use the `Gestalt` function to determine whether these features are available. Call the `Gestalt` function with the `gestaltResourceMgrAttr` selector to determine whether the routines for handling partial resources exist. If the bit indicated by the constant `gestaltPartialRsrcs` is set, then the Resource Manager routines for handling partial resources are available. For more information about the `Gestalt` function, see *Inside Macintosh: Operating System Utilities*.

```
CONST
   gestaltResourceMgrAttr  = 'rsrc';   {Gestalt selector for }
                                       { Resource Mgr attributes}
   gestaltPartialRsrcs     = 0;        {check this bit in the }
                                       { response parameter}
```

You can use the `ResError` function to retrieve errors that may result from calling Resource Manager routines. Resource Manager procedures do not report error information directly. Instead, after calling a Resource Manager procedure your application should call the `ResError` function to determine whether an error occurred.

Resource Manager functions usually return `NIL` or –1 as the function result when there's an error. For Resource Manager functions that return –1, your application can call the `ResError` function to determine the specific error that occurred. For Resource Manager functions that return handles, your application should always check whether the value of the returned handle is `NIL`. If it is, your application can use `ResError` to obtain specific information about the nature of the error. Note, however, that in some cases `ResError` returns `noErr` even though the value of the returned handle is `NIL`.

The rest of this section describes how to create a resource using ResEdit or the Rez resource compiler. It then describes how to use Resource Manager routines to

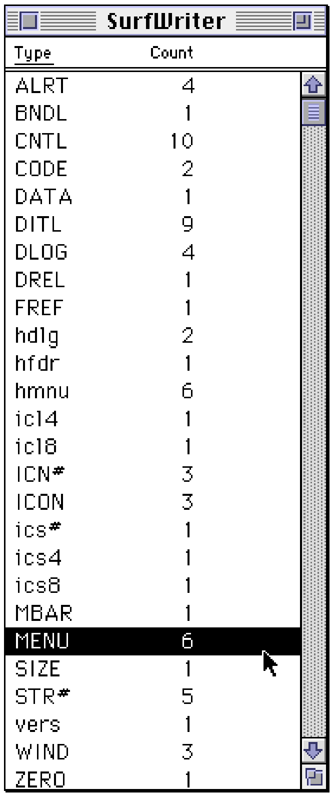
- get a handle to a resource and modify a purgeable resource safely
- release and detach resources
- create and open a resource fork
- set the current resource file (the file whose resource fork the Resource Manager searches first)
- read and manipulate resources
- write resources
- read and write partial resources

For detailed descriptions of all Resource Manager routines, see "Resource Manager Reference" beginning on page 1-42. For information on writing data to a file's data fork, see *Inside Macintosh: Files*.

## Creating a Resource

You typically define the user interface elements of your application, such as menus, windows, dialog boxes, and controls, by specifying descriptions of these elements in resources. You can then use Menu Manager, Window Manager, Dialog Manager, or Control Manager routines to create these elements—based on their resource descriptions—as needed. You can create resource descriptions using a resource editor, such as ResEdit, which lets you create the resources in a visual manner; or you can provide a textual, formal description of resources in a file and then use a resource compiler, such as Rez, to compile the description into a resource. Figure 1-5 shows the window ResEdit displays for the SurfWriter application. This window lists all of the resources in the resource fork of the SurfWriter application.

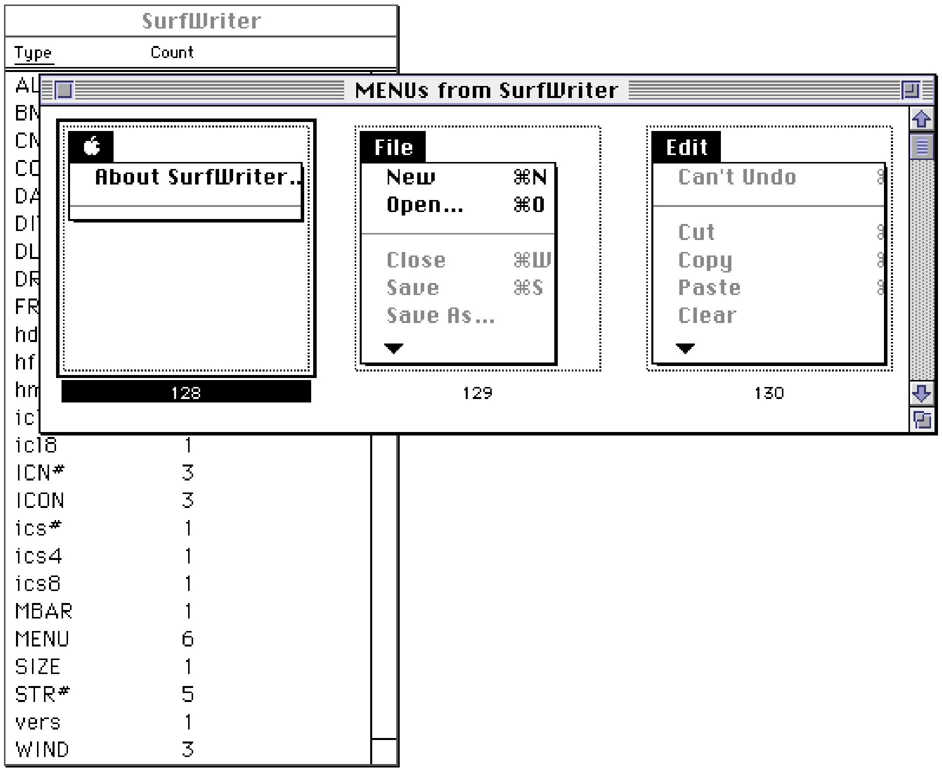
**Figure 1-5** The ResEdit window for the SurfWriter application



| Type | Count |
|---|---|
| ALRT | 4 |
| BNDL | 1 |
| CNTL | 10 |
| CODE | 2 |
| DATA | 1 |
| DITL | 9 |
| DLOG | 4 |
| DREL | 1 |
| FREF | 1 |
| hdlg | 2 |
| hfdr | 1 |
| hmnu | 6 |
| icl4 | 1 |
| icl8 | 1 |
| ICN# | 3 |
| ICON | 3 |
| ics# | 1 |
| ics4 | 1 |
| ics8 | 1 |
| MBAR | 1 |
| MENU | 6 |
| SIZE | 1 |
| STR# | 5 |
| vers | 1 |
| WIND | 3 |
| ZERO | 1 |

You can use ResEdit to examine any of your application's resources. For example, to view your application's `'MENU'` resources, double-click that resource in the ResEdit window. Figure 1-6 shows how ResEdit displays the menus of the SurfWriter application.

**Figure 1-6** The menus of the SurfWriter application



| Type | Count |
|---|---|
| AL | |
| BN | |
| CN | |
| CO | |
| DA | |
| DI | |
| DL | |
| DR | |
| FR | |
| hd | |
| hf | |
| hm | |
| ic | |
| icl8 | 1 |
| ICN# | 3 |
| ICON | 3 |
| ics# | 1 |
| ics4 | 1 |
| ics8 | 1 |
| MBAR | 1 |
| MENU | 6 |
| SIZE | 1 |
| STR# | 5 |
| vers | 1 |
| WIND | 3 |

Listing 1-1 shows the definition of SurfWriter's Apple menu in Rez input format.

**Listing 1-1** A menu in Rez input format

```
#define mApple 128

resource 'MENU' (mApple, preload) { /*resource ID, preload resource*/
      mApple,                       /*menu ID*/
      textMenuProc,                 /*uses standard menu definition */
                                    /* procedure*/
      0b1111111111111111111111111111101,  /*enable About item, */
                                          /* disable divider, */
                                          /* enable all other items*/
      enabled,                      /*enable menu title*/
      apple,                        /*menu title*/
      {
                                    /*first menu item*/
            "About SurfWriter…",                /*text of menu item*/
                  noicon, nokey, nomark, plain; /*item characteristics*/
                                    /*second menu item*/
            "-",                                /*item text (divider)*/
                  noicon, nokey, nomark, plain  /*item characteristics*/
      }
};
```

Your application can also create, modify, and save resources as needed using various Resource Manager routines.

You can store your application-specific resources in the application file itself. You need not add resources to your application after it is created. Instead, store any document-specific resources in the relevant document and store user preferences in a preferences file in the Preferences folder of the System Folder.

To retrieve resources from your application's resource fork, you usually use other managers (such as the Menu Manager or Window Manager). To retrieve resources other than menus, windows, dialog boxes, or controls, you usually use Resource Manager routines.

To retrieve a resource from a document file or a preferences file, your application needs to open the resource fork of the file and then use Resource Manager routines to retrieve any resources in the file. The section that follows, “Getting a Resource,” describes how the Resource Manager returns a handle to a resource at your application’s request and how to modify a purgeable resource safely. The sections “Opening a Resource Fork” and “Reading and Manipulating Resources” beginning on page 1-24 and page 1-30, respectively, describe in detail how to use Resource Manager routines to open and read resources.

## Getting a Resource

You usually use the `GetResource` function to read data from resources other than menus, windows, dialog boxes, and controls. You supply the resource type and resource ID of the desired resource, and the `GetResource` function searches the resource maps of open resource forks (according to the search path described in “Search Path for Resources” beginning on page 1-10) for that resource’s entry.

If the `GetResource` function finds an entry for the requested resource in the resource map and the resource is in memory (that is, if the resource map in memory does not specify the resource’s location with a handle whose value is `NIL`), `GetResource` returns a handle to the resource. If the resource is listed in the resource map but is not in memory (the resource map in memory specifies the resource’s location with a handle whose value is `NIL`), `GetResource` reads the resource data from disk into memory, replaces the entry for the resource’s location with a handle to the resource, and returns to your application a handle to the resource. For a resource that cannot be purged (that is, whose purgeable attribute is not set) you can use the returned handle to refer to the resource in other Resource Manager routines. (Handles to purgeable resources are discussed later in this section.)

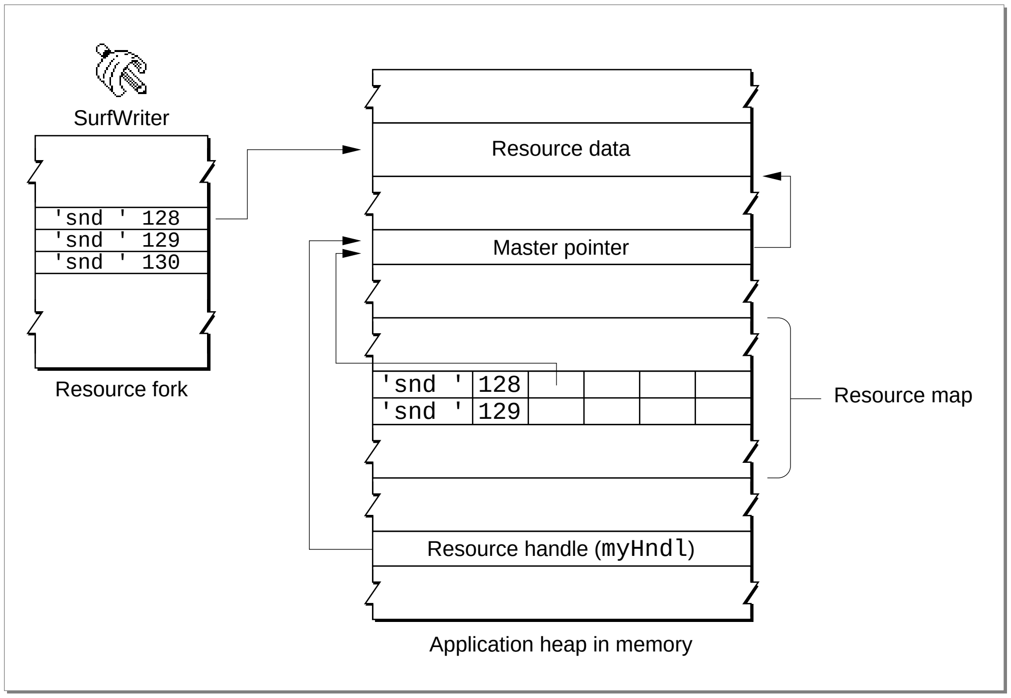
For example, this code uses `GetResource` to get a handle to an `'snd '` resource with resource ID 128.

```
VAR
   resourceType:  ResType;
   resourceID:    Integer;
   myHndl:        Handle;

resourceType := 'snd ';
resourceID := 128;
myHndl := GetResource(resourceType, resourceID);
```

Figure 1-7 shows how `GetResource` returns a handle to a resource at your application's request.

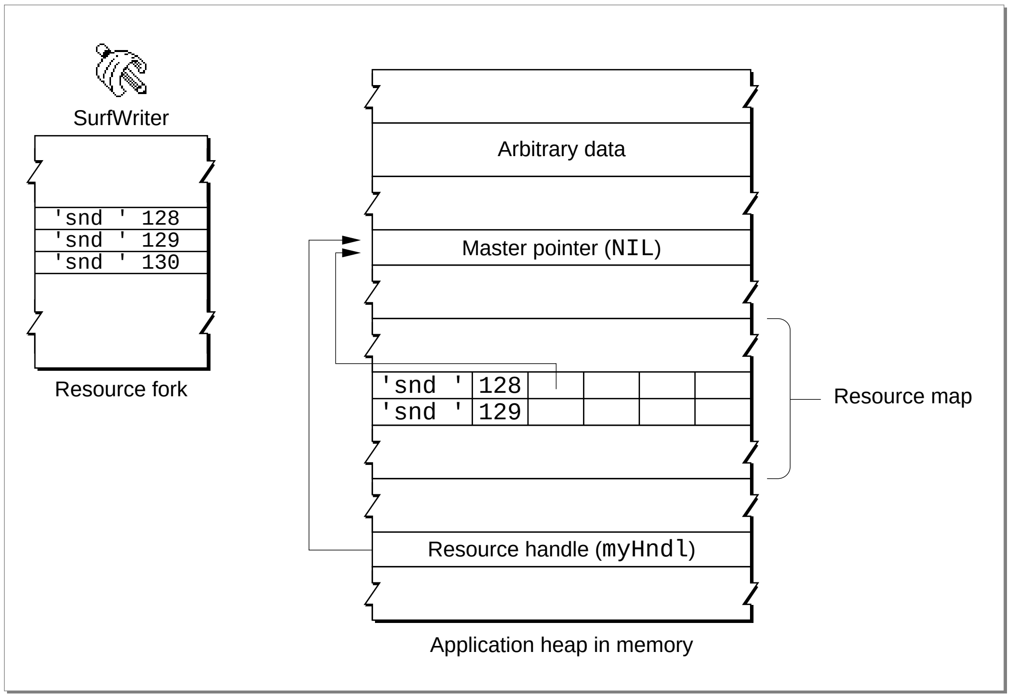
**Figure 1-7** Getting a handle to a resource



Note that the handle returned to your application is a copy of the handle in the resource map. The resource map contains a handle to the resource data, and the Resource Manager returns a handle to the same block of memory for use by your application. If you use `GetResource` to get a handle to a resource that has the purgeable attribute set or if you intend to modify such a resource, keep the following discussion in mind.

If a resource is marked purgeable and the Memory Manager determines that it must purge a resource to make more room in your application's heap, it releases the memory occupied by the resource. In this case, the handle to the resource data is no longer valid, because the handle's master pointer is set to `NIL`. If your application attempts to use the handle previously returned by the Resource Manager, the handle no longer refers to the resource. Figure 1-8 shows a handle to a resource that is no longer valid, because the Memory Manager has purged the resource. To avoid this situation, you should call the `LoadResource` procedure to make sure that the resource is in memory before attempting to refer to it.

**Figure 1-8** A handle to a purgeable resource after the resource has been purged



If you need to make changes to a purgeable resource using routines that may cause the Memory Manager to purge the resource, you should make the resource temporarily not purgeable. You can use the Memory Manager procedures `HGetState`, `HNoPurge`, and `HSetState` for this purpose. After calling `HGetState` and `HNoPurge`, change the resource as necessary. To make the changes permanent, use the `ChangedResource` and `WriteResource` procedures; then call `HSetState` when you're finished. Listing 1-2 illustrates the use of these routines.

**Listing 1-2** Safely changing a resource that is purgeable

```
VAR
   resourceType:  ResType;
   resourceID:    Integer;
   myHndl:        Handle;
   state:         SignedByte;

resourceType := 'snd ';
resourceID := 128;
{read the resource into memory}
myHndl := GetResource(resourceType, resourceID);
state := HGetState(myHndl); {get the state of the handle}
HNoPurge(myHndl);          {mark the handle as not purgeable}
{modify the resource as needed}
{...}
ChangedResource(myHndl);   {mark the resource as changed}
WriteResource(myHndl);     {write the resource to disk}
HSetState(myHndl, state);  {restore the handle's state}
```

Although you'll usually want to use `WriteResource` to write a resource's data to disk immediately (as shown in Listing 1-2), you can instead use the `SetResPurge` procedure and specify `TRUE` in the `install` parameter. If you do this, the Memory Manager calls the Resource Manager before purging data specified by a handle. The Resource Manager determines whether the passed handle is that of a resource in your application's heap, and, if so, calls `WriteResource` to write the resource to disk if its changed attribute is set. You can call the `SetResPurge` procedure and specify `FALSE` in the `install` parameter to restore the normal state, so that the Memory Manager purges resource data in memory without checking with the Resource Manager.

## Releasing and Detaching Resources

When you've finished using a resource, you can call `ReleaseResource` to release the memory associated with that resource. For a given resource, the `ReleaseResource` procedure releases the memory associated with the resource, setting the handle's master pointer to `NIL`, thus making your application's handle to the resource invalid. (This is similar to the situation shown in Figure 1-8.) After releasing a resource, use another Resource Manager routine if you need to use the resource again. For example, the code in Listing 1-3 first uses `GetResource` to get a handle to a resource, manipulates the resource, then uses `ReleaseResource` when the application has finished using the resource. If the application needs the resource later, it must get a valid handle to the resource by reading the resource into memory again (using `GetResource`, for example).

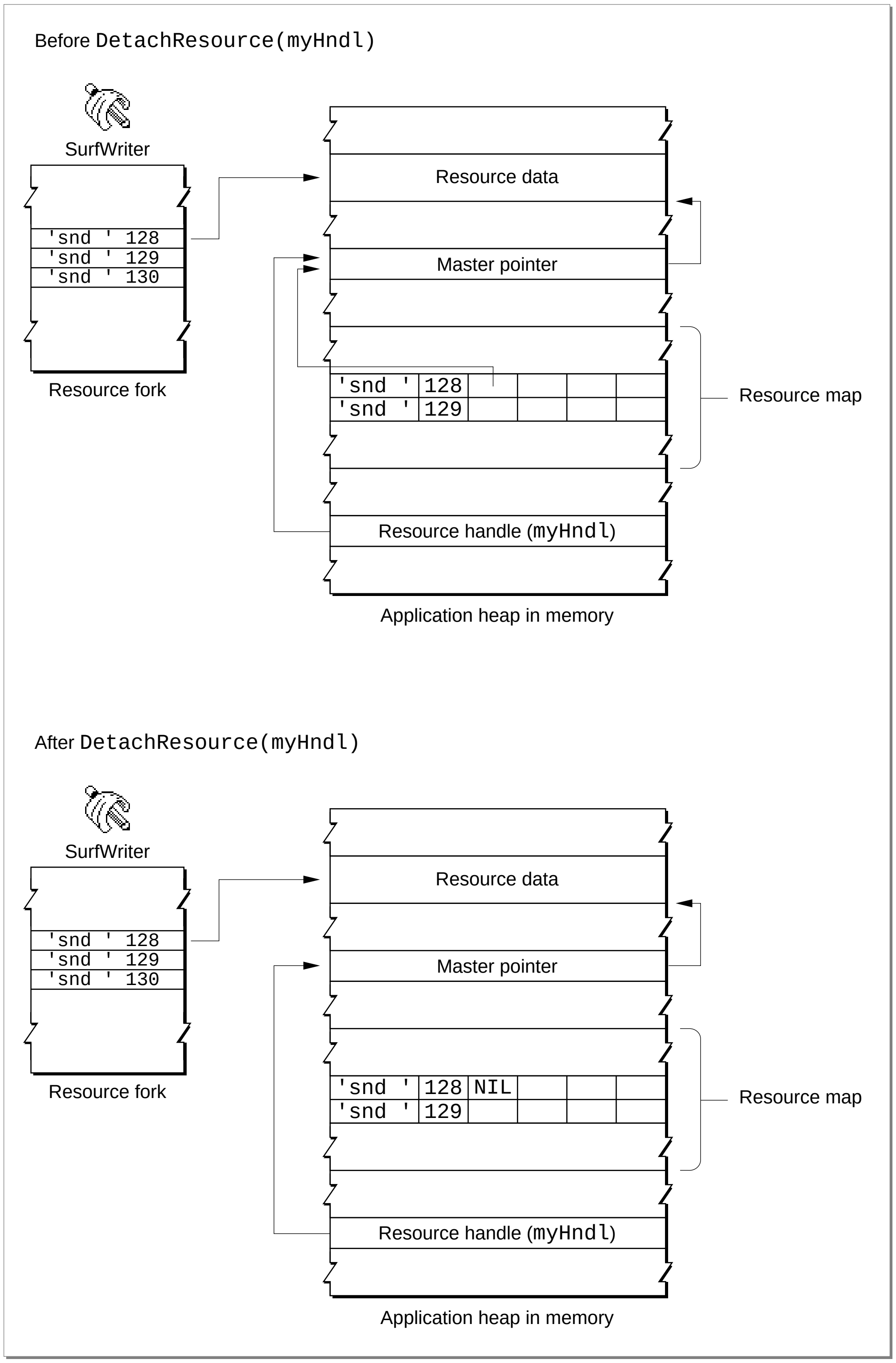
**Listing 1-3** Releasing a resource

```
PROCEDURE MyGetAndPlaySoundResource(resourceID: Integer);
VAR
   myHndl: Handle;
BEGIN
   myHndl := GetResource('snd ', resourceID);
   {use the resource}
   {when done, release the resource}
   ReleaseResource(myHndl);
END;
```

Your application can also use the `DetachResource` procedure to replace a resource's handle in the resource map with a handle whose value is `NIL`. However, the `DetachResource` procedure does not release the memory associated with the resource. You can use `DetachResource` when you want your application to access the resource's data directly, without the aid of the Resource Manager, or when you need to pass the handle to a routine that does not accept a resource handle. (For example, the `AddResource` routine used in Listing 1-4 on page 1-24 takes a handle to data, not a handle to a resource.) Once you detach a resource, the Resource Manager does not recognize the resource's handle in the resource map in memory as a valid handle to a resource, but your application can still manipulate the resource's data through its own handle to the data.

Figure 1-9 shows how both your application and the Resource Manager have a handle to a resource after your application calls `GetResource`. The figure also shows how the Resource Manager replaces the handle in the resource map in memory with a handle whose value is `NIL` when your application calls `DetachResource`.

**Figure 1-9** Detaching a resource



You can also easily copy a resource by first reading in the resource using `GetResource`, detaching the resource using `DetachResource`, then copying the resource by using `AddResource` (and specifying a new resource ID). Listing 1-4 uses this technique to copy a resource within the current resource file.

**Listing 1-4** Detaching a resource

```
PROCEDURE MyCopyAResource(resourceType: ResType;
                           resourceID: Integer;
                           VAR myHndl: Handle);
VAR
   newResourceID: Integer;
BEGIN
   myHndl := GetResource(resourceType, resourceID);
   DetachResource(myHndl);                {detach the resource}
   newResourceID := UniqueID(resourceType);
   AddResource(myHndl, resourceType, newResourceID, '');
END;
```

## Opening a Resource Fork

When your application opens a file's resource fork or data fork, the File Manager returns a file reference number. You use a file reference number in File Manager routines (and in a few Resource Manager routines) to identify a unique access path to an open fork of a specific file. Even though the file reference number of the data fork and the resource fork usually match, you should use the file reference number of a file's resource fork in Resource Manager routines; don't assume that it has the same value as the file reference number for the same file's data fork.

### Opening an Application's Resource Fork

Because system software automatically opens your application's resource fork when the user opens your application, you do not need to open it explicitly. However, you should save your application's file reference number. You can do this by calling the `CurResFile` function early in your initialization procedure. (The `CurResFile` function returns the file reference number of the current resource file.) Listing 1-5 shows the part of SurfWriter's initialization procedure that gets the file reference number of the application's resource fork.

**Listing 1-5** Getting the file reference number for your application's resource fork

```
PROCEDURE MyInitialize;
BEGIN
   MaxApplZone;         {extend heap zone to limit}
   MoreMasters;         {get 64 more master pointers}
   MoreMasters;         {get 64 more master pointers}
   InitGraf(@thePort);  {initialize QuickDraw}
   InitFonts;           {initialize Font Manager}
   InitWindows;         {initialize Window Manager}
   TEInit;              {initialize TextEdit}
   InitDialogs(Nil);    {initialize Dialog Manager}
   InitCursor;          {set cursor to arrow}
   {get the file ref num of this app's resource file }
   { and save it in a global variable}
   gAppsResourceFork := CurResFile;
   {do other initialization}
END;
```

SurfWriter uses an application-defined global variable (`gAppsResourceFork`) to refer to its resource fork in subsequent calls to Resource Manager routines.

## Creating and Opening a Resource Fork

To save resources in the resource fork of a file, you must first create the resource fork (if it doesn't already exist in a form that can be used by the Resource Manager) and obtain a file reference number for it. After creating a new resource fork, you must open it before writing any resources to it. You'll usually want to save the file reference number of any resource fork that your application opens.

To create a resource fork, use the `FSpCreateResFile` procedure. This procedure requires four parameters: a file-system specification record (identifying the name and location of the file), the signature of the application creating the file, the file type of the file, and the script code for the file.

A file system specification record is a standard format for identifying a file or directory. The file system specification record for files and directories is available in System 7 and later versions of system software and is defined by the `FSSpec` data type.

```
TYPE FSSpec = {file system specification}
   RECORD
      vRefNum: Integer;    {volume reference number}
      parID:   LongInt;    {directory ID of parent directory}
      name:    Str63;      {filename or directory name}
   END;
```

Certain File Manager routines—those that open a file's data fork—also take a file system specification record as a parameter. You can use the same `FSSpec` record in Resource Manager routines that create or open the file's resource fork.

If the file specified by the `FSSpec` record doesn't already exist (that is, if the file has neither a data fork nor a resource fork), the `FSpCreateResFile` procedure creates a resource file—that is, a resource fork, including a resource map. In this case, the file has a zero-length data fork. The `FSpCreateResFile` procedure also sets the creator, type, and script code fields of the file's catalog information record to the specified values.

If the file specified by the `FSSpec` record already exists and includes a resource fork with a resource map, `FSpCreateResFile` does nothing, and the `ResError` function returns an appropriate result code. If the data fork of the file specified by the `FSSpec` record already exists but the file has a zero-length resource fork, `FSpCreateResFile` creates an empty resource fork and resource map for the file; it also changes the creator, type, and script code fields of the catalog information record of the file to the specified values.

Listing 1-6 shows a function that creates a new resource fork, including a resource map.

**Listing 1-6** Creating an empty resource fork

```
FUNCTION MyCreateResourceFork (myFSSpec: FSSpec): OSErr;
BEGIN
   FSpCreateResFile(myFSSpec, gAppSignature, 'TEXT',
                    smSystemScript);
   MyCreateResourceFork := ResError;
END;
```

After creating a resource fork, you can open it using the `FSpOpenResFile` function. The `FSpOpenResFile` function returns a file reference number that you can use to change or limit the Resource Manager's search order or to close a resource fork.

After opening a resource fork, you can write resources to it using the routines described in "Writing Resources" beginning on page 1-36. (You can also write to a resource fork using File Manager routines; in general, you should use the Resource Manager.) When you are finished using a resource fork that your application has specifically opened, you should close it using the `CloseResFile` procedure. The Resource Manager automatically closes any resource forks opened by your application that are still open when your application calls `ExitToShell`.

Listing 1-7 shows a routine that uses the application-defined function `MyCreateResourceFork` (shown in Listing 1-6) to create a new resource fork, opens the resource fork, writes resources to it, then closes the resource fork when it is finished.

**Listing 1-7** Creating and opening a resource fork

```
FUNCTION MyCreateAndOpenResourceFork (myFSSpec: FSSpec): OSErr;
VAR
   myErr:      OSErr;
   myRefNum:   Integer;
BEGIN
                                 {create a resource file}
   myErr := MyCreateResourceFork(myFSSpec);
   IF myErr = noErr THEN         {open the resource file}
      myRefNum := FSpOpenResFile(myFSSpec, fsRdWrPerm);
   IF ResError = noErr THEN      {write to the resource file}
      myErr := MyWriteResourcesToFile(myRefNum);
   CloseResFile(myRefNum);       {close the resource file}
   MyCreateAndOpenResourceFork := myErr;
END;
```

Note that when you open a resource fork, the Resource Manager resets the search path so that the file whose resource fork you just opened becomes the current resource file. For example, suppose the SurfWriter application file is open, and the user opens document A, then document B. SurfWriter opens the resource forks of both documents. In this case, the search order is

1. document B (the current resource file)
2. document A
3. the SurfWriter application
4. the System file

If the user is working with document A and SurfWriter uses the `UseResFile` procedure to set the current resource file to document A, the new search order is

1. document A (the current resource file)
2. the SurfWriter application
3. the System file

If the user opens another document, document C, and SurfWriter opens its resource fork, the new search order becomes

1. document C (the current resource file)
2. document B
3. document A
4. the SurfWriter application
5. the System file

## Specifying the Current Resource File

When you request a resource, the Resource Manager follows the search order described in "Search Path for Resources" on page 1-10. To change the starting point of the search or to restrict the search to the resource fork of a specific file, you can use `CurResFile` and `UseResFile`. To get the file reference number for the current resource file, use the `CurResFile` function. You can then use the `UseResFile` procedure to set the current resource file to the desired file, use other Resource Manager routines to retrieve any desired resources, and then use `UseResFile` again to restore the current resource file to its previous setting.

For example, the SurfWriter application allows users to specify or record either a special reward sound that applies only to a specific document or a general reward sound that can apply to any document. SurfWriter stores a document-specific reward sound resource in the document and the general reward sound resource in either the SurfWriter Preferences file (if the reward sound is user-defined) or in the application file. If several documents are open and SurfWriter needs to play a document-specific reward sound, SurfWriter attempts to get the sound from that document without searching the resource forks of any other documents that might be open. If the document doesn't have the specified reward sound, SurfWriter searches for the sound in the resource fork of the preferences file and, if necessary, of the application file and System file.

Listing 1-8 shows how the SurfWriter application uses `CurResFile` and `UseResFile` to get and play the appropriate reward sound for a given document. All reward sounds share the same resource ID, `kProductiveWriter`. The application-defined procedure `MyGetAndPlayRewardSoundResource` first checks whether the reward sound setting for the document specifies a sound stored in that document or a general reward sound stored in the preferences file or elsewhere. If the document has a reward sound, the procedure sets the current resource file to that document, searches just that document's resource fork for the sound, and plays the sound. If the document doesn't have a reward sound, the `MyGetAndPlayRewardSoundResource` procedure sets the current resource file to SurfWriter Preferences, searches the entire resource chain from that point on for the sound, and plays the sound. This scheme ensures that SurfWriter always plays the correct reward sound, even if different reward sound resources in different documents share the same resource ID.

**Listing 1-8** Saving and restoring the current resource file

```
PROCEDURE MyGetAndPlayRewardSoundResource (refNum: Integer);
VAR
   myHndl:        Handle;
   prevResFile:   Integer;
BEGIN
   prevResFile := CurResFile; {save the current resource file}
   IF MyHasDocumentRewardSound(refNum) THEN
   BEGIN
      {first set the current resource file to a specific document}
      UseResFile(refNum);
      {get reward sound from the document using Get1Resource }
      { to limit search to current resource file and avoid }
      { searching the resource forks of any other open documents}
      myHndl := Get1Resource('snd ', kProductiveWriter);
   END
   ELSE
   BEGIN
      {set current resource file to SurfWriter Preferences}
      UseResFile(gSurfPrefsResourceFork);
      {get reward sound resource using GetResource to search }
      { entire resource chain starting with preferences file}
      myHndl := GetResource('snd ', kProductiveWriter);
   END;
   IF myHndl <> NIL THEN
   BEGIN
      MyPlayThisSound(myHndl);
      ReleaseResource(myHndl);
   END;
   UseResFile(prevResFile);{restore the current resource }
                           { file to its previous setting}
END;
```

The MyGetAndPlayRewardSoundResource procedure saves the reference number of the current resource file and then calls the application-defined routine MyHasDocumentRewardSound to check whether the document has a reward sound associated with it. If so, MyGetAndPlayRewardSoundResource sets the current resource file to the value specified by the refNum parameter. The procedure then uses the Get1Resource function to get, from the current resource file, a handle to the resource of type 'snd ' with the ID specified by kProductiveWriter.

If the document doesn't have a specified reward sound, MyGetAndPlayRewardSoundResource uses UseResFile to set the current resource file to the SurfWriter Preferences file's resource fork and GetResource to search the entire resource chain from that point. If GetResource locates a resource with the specified resource ID in the SurfWriter Preferences file, it returns a handle to that resource; if not, it continues to search until it finds the specified resource or reaches the end of the resource chain. This ensures that the procedure won't get a user-defined resource with the same resource ID in some other SurfWriter document that is currently open instead of the general reward sound saved in SurfWriter Preferences or in SurfWriter itself.

If the call to Get1Resource or GetResource is successful (that is, if it does not return a handle whose value is NIL), MyGetAndPlayRewardSoundResource plays the appropriate reward sound, then uses ReleaseResource to release the memory occupied by the sound resource. Finally, the procedure uses UseResFile to restore the current resource file to its previous setting.

## Reading and Manipulating Resources

The Resource Manager provides a number of routines that read resources from a resource fork. When you request a resource, the Resource Manager follows the search path described in "Search Path for Resources" on page 1-10. That is, the Resource Manager searches each resource fork open to your application, beginning with the current resource file, and continues until it either finds the resource or reaches the end of the chain.

You can change where the Resource Manager starts its search using the UseResFile procedure. (See the previous section, "Specifying the Current Resource File," for details.) You can limit the search to only the current resource file by using the Resource Manager routines that contain a "1" in their names, such as Get1Resource, Get1NamedResource, Get1IndResource, Unique1ID, and Count1Resources.

To get a resource, you can specify it by its resource type and resource ID or by its resource type and resource name. By convention, most applications refer to a resource by its resource type and resource ID, rather than by its resource type and resource name.

You can use the `SetResLoad` procedure to enable and disable automatic loading of resource data into memory for routines that return handles to resources. Such routines normally read the resource data into memory if it's not already there. This is the default setting and the effect of calling `SetResLoad` with the `load` parameter set to `TRUE`. If you call `SetResLoad` with the `load` parameter set to `FALSE`, subsequent calls to routines that return handles to resources will not load the resource data into memory. Instead, such routines return a handle whose master pointer is set to `NIL` unless the resource is already in memory. This setting is useful when you want to read from the resource map without reading the resource data into memory. To read the resource data into memory after a call to `SetResLoad` with the `load` parameter set to `FALSE`, call `LoadResource`.

▲ **WARNING**
If you call `SetResLoad` with the `load` parameter set to `FALSE`, be sure to call `SetResLoad` with the `load` parameter set to `TRUE` as soon as possible. Other parts of system software that call the Resource Manager rely on the default setting (the `load` parameter set to `TRUE`), and some routines won't work if resources are not loaded automatically. ▲

In addition to the `SetResLoad` procedure, you can use the preloaded attribute of an individual resource to control loading of that resource's data into memory. The Resource Manager loads a resource into memory when it first opens a resource fork if the resource's preloaded attribute is set.

**Note**
If both the preloaded attribute and the locked attribute are set, the Resource Manager loads the resource as low in the heap as possible. ◆

Here's an example of a situation in which an application might need to read a resource. The SurfWriter application always saves the last position of a document window when the user saves the document, storing this information in a resource that it has defined for this purpose. SurfWriter defines a resource with resource type `rWinState` and resource ID `kLastWinStateID` to store information about the window (its position and its state—that is, either the user state or the standard state). SurfWriter's window state resource has this format, defined by a record of type `MyWindowState`:

```
TYPE MyWindowState =
   RECORD
      userStateRect: Rect;    {user state rectangle}
      zoomState:     Boolean; {window state: TRUE = standard; }
                              {              FALSE = user}
   END;

   MyWindowStatePtr = ^MyWindowState;
   MyWindowStateHnd = ^MyWindowStatePtr;
```

Listing 1-9 shows a procedure called `MySetWindowPosition` that the SurfWriter application uses in the process of opening a document. The SurfWriter application stores the last location of a document in its window state resource. When SurfWriter opens the document again, it uses `MySetWindowPosition` to read the document's window state resource and uses the resource data to set the window's location.

**Listing 1-9** Getting a resource from a document file

```
PROCEDURE MySetWindowPosition (myWindow: WindowPtr);
VAR
   myData:              MyDocRecHnd;
   lastUserStateRect:   Rect;
   stdStateRect:        Rect;
   curStateRect:        Rect;
   myRefNum:            Integer;
   myStateHandle:       MyWindowStateHnd;
   resourceGood:        Boolean;
   savePort:            GrafPtr;
   myErr:               OSErr;
BEGIN
   myData := MyDocRecHnd(GetWRefCon(myWindow));    {get document record}
   HLock(Handle(myData));        {lock the record while manipulating it}
   {open the resource fork and get its file reference number}
   myRefNum := FSpOpenResFile(myData^^.fileFSSpec, fsRdWrPerm);
   myErr := ResError;
   IF myErr <> noErr THEN
      Exit(MySetWindowPosition);
   {get handle to rectangle that describes document's last window position}
   myStateHandle := MyWindowStateHnd(Get1Resource(rWinState,
                                                kLastWinStateID));
   IF myStateHandle <> NIL THEN                {handle to data succeeded}
   BEGIN    {retrieve the saved user state}
      lastUserStateRect := myStateHandle^^.userStateRect;
      resourceGood := TRUE;
   END
   ELSE
   BEGIN
      lastUserStateRect.top := 0;   {force MyVerifyPosition to calculate }
      resourceGood := FALSE;        { the default position}
   END;
   {verify that user state is practical and calculate new standard state}
   MyVerifyPosition(myWindow, lastUserStateRect, stdStateRect);
   IF resourceGood THEN                   {document had state resource}
      IF myStateHandle^^.zoomState THEN   {if window was in standard state }
         curStateRect := stdStateRect     { when saved, display it in }
                                          { newly calculated standard state}
      ELSE                 {otherwise, current state is the user state}
         curStateRect := lastUserStateRect
   ELSE                                   {document had no state resource}
      curStateRect := lastUserStateRect;  {use default user state}
   {move window}
   MoveWindow(myWindow, curStateRect.left, curStateRect.top, FALSE);
   {convert to local coordinates and resize window}
   GetPort(savePort);
   SetPort(myWindow);
   GlobalToLocal(curStateRect.topLeft);
   GlobalToLocal(curStateRect.botRight);
   SizeWindow(myWindow, curStateRect.right, curStateRect.bottom, TRUE);
   IF resourceGood THEN    {reset user state and standard }
   BEGIN                   { state--SizeWindow may have changed them}
      MySetWindowUserState(myWindow, lastUserStateRect);
      MySetWindowStdState(myWindow, stdStateRect);
   END;
   ReleaseResource(Handle(myStateHandle));        {clean up}
   CloseResFile(myRefNum);
   HUnlock(Handle(myData));
   SetPort(savePort);
END;
```

The `MySetWindowPosition` procedure uses the `FSpOpenResFile` function to open the document's resource fork, then uses `Get1Resource` to get a handle to the resource that contains information about the window's last position. The procedure can then verify that the saved position is practical and move the window to that position.

Note that when a Resource Manager routine returns a handle to a resource, the routine returns the resource using the `Handle` data type. You usually define a data type (such as `MyWindowState`) to access the resource's data. If you also define a handle to your defined data type (such as `MyWindowStateHnd`), you need to coerce the returned handle to the appropriate type, as shown in this line from Listing 1-9:

```
myStateHandle := MyWindowStateHnd(Get1Resource(rWinState, kLastWinStateID));
```

If you use this method, you also need to coerce your defined handle back to a handle of type `Handle` when you use other Resource Manager routines. For example, after it has finished moving the window, `MySetWindowPosition` uses `ReleaseResource` to release the memory allocated to the resource's data (which also sets the master pointer of the resource's handle in the resource map in memory to `NIL`). As shown in this line from Listing 1-9, SurfWriter coerces the defined handle back to a handle:

```
ReleaseResource(Handle(myStateHandle));
```

After releasing the resource data's memory, `MySetWindowPosition` uses the `CloseResFile` procedure to close the resource fork.

**Note**
Listing 1-9 assumes the window state resource is not purgeable. If it were, `MySetWindowPosition` would need to call `LoadResource` before accessing the data in the resource. ◆

The Resource Manager also provides routines that let you index through all resources of a given type (for example, using `CountResources` and `GetIndResource`). This can be useful whenever you want to read all the resources of a given type.

Listing 1-10 shows an application-defined procedure that allows a user to open a file that contains sound resources. The SurfWriter application opens the specified file, counts the number of `'snd '` resources in the file, then performs an operation on each `'snd '` resource (adding the name of each resource to its Sounds menu).

**Listing 1-10** Counting and indexing through resources

```
PROCEDURE MyDoOpenSoundResources;
VAR
   mySFReply:     StandardFileReply;{reply record}
   myNumTypes:    Integer;          {number of types to display}
   myTypeList:    SFTypeList;       {file type of files}
   myRefNum:      Integer;          {resource file reference no}
   mySndHandle:   Handle;           {handle to sound resource}
   numberOfSnds:  Integer;          {# of sounds in resource file}
   index:         Integer;          {index of sound resource}
   resName:       Str255;           {name of sound resource}
   curRes:        Integer;          {saved current resource file}
   myType:        ResType;          {resource type}
   myResID:       Integer;          {resource ID of snd resource}
   myWindow:      WindowPtr;        {window pointer}
   menu:          MenuHandle;       {handle to Sounds menu}
   myErr:         OSErr;            {error information}
```

```
BEGIN
   curRes := CurResFile;
   myWindow := FrontWindow;
   MyDoActivate(myWindow, FALSE);   {deactivate front window}
   myTypeList[0] := 'SFSD';         {show files of this type}
   myNumTypes := 1;
   {let user choose a file that contains sound resources}
   StandardGetFile(NIL, myNumTypes, myTypeList, mySFReply);
   IF mySFReply.sfGood = TRUE THEN
   BEGIN
      myRefNum := FSpOpenResFile(mySFReply.sfFile, fsRdWrPerm);
      IF myRefNum = -1 THEN
         DoError;
      menu := GetMenuHandle(mSounds);
      numberOfSnds := Count1Resources('snd ');
      FOR index := 1 TO numberOfSnds DO
      BEGIN {the loop}
         mySndHandle := Get1IndResource('snd ', index);
         IF mySndHandle = NIL THEN
            DoError
         ELSE
         BEGIN
            GetResInfo(mySndHandle, myResID, myType, resName);
            AppendMenu(menu, resName);
            ReleaseResource(mySndHandle);
         END;  {of mySndHandle <> NIL}
      END;  {of the loop}
   UseResFile(curRes);
   gSoundResFileRefNum := myRefNum;
   END;  {of sfReply.good}
END;
```

After the user selects a file that contains SurfWriter sound resources (that is, a file of type `'SFSD'`), the `MyDoOpenSoundResources` procedure calls `FSpOpenResFile` to open the file's resource fork and obtain its file reference number. (If `FSpOpenResFile` fails to open the resource fork, it returns –1 instead of a file reference number.) The `MyDoOpenSoundResources` procedure then uses the `Count1Resources` function to count the number of `'snd '` resources in the resource fork. It can then index through the resources one at a time, using `Get1IndResource` to open each resource, `GetResInfo` to get the resource's name, and `AppendMenu` to append each name to SurfWriter's Sounds menu.

**Note**
In most situations, you can use the Menu Manager procedure `AppendResMenu` to add names of resources to a menu. See *Inside Macintosh: Macintosh Toolbox Essentials* for details. ◆

## Writing Resources

After opening a resource fork (as described in "Creating and Opening a Resource Fork" beginning on page 1-25), you can write resources to it. You can write resources only to the current resource file. To ensure that the current resource file is set to the appropriate resource fork, you can use `CurResFile` to save the file reference number of the current resource file, then `UseResFile` to set the current resource file to the desired resource fork.

To specify data for a new resource, you usually use the `AddResource` procedure, which creates a new entry for the resource in the resource map in memory and sets the entry's location to refer to the resource's data. Note that `AddResource` changes only the resource map in memory; it doesn't change anything on disk. Use the `UpdateResFile` or `WriteResource` procedure to write the resource to disk. The `AddResource` procedure always adds the resource to the resource map in memory that corresponds to the current resource file. For this reason, you usually need to set the current resource file to the desired file before calling `AddResource`.

If you change a resource that is referenced through the resource map in memory, you use the `ChangedResource` procedure to set the `resChanged` attribute of that resource's entry. You should then immediately call the `UpdateResFile` or `WriteResource` procedure to write the changed resource data to disk. Note that although the `UpdateResFile` procedure writes only those resources that have been added or changed to disk, it also writes the entire resource map to disk (overwriting its previous contents). The `WriteResource` procedure writes only the resource data of a single resource to disk; it does not update the resource's entry in the resource map on disk.

The `ChangedResource` procedure reserves enough disk space to contain the changed resource. It does this every time it's called, but the actual writing of the resource does not take place until a call to `WriteResource` or `UpdateResFile`. Thus, if you call `ChangedResource` several times on a large resource before the resource is actually written, you may unexpectedly run out of disk space, because many times the amount of space actually needed is reserved. When the resource is actually written, the file's end-of-file (`EOF`) is set correctly, and the next call to `ChangedResource` will work as expected.

**IMPORTANT**

If your application frequently changes the contents of resources (especially large resources), you should call `WriteResource` or `UpdateResFile` immediately after calling `ChangedResource`. ▲

To ensure that the Resource Manager does not purge a purgeable resource while your application is in the process of changing it, use the Memory Manager procedures `HGetState`, `HNoPurge`, and `HSetState`. First call `HGetState` and `HNoPurge`, then change the resource as necessary. To make a change to a resource permanent, use the `ChangedResource` and `WriteResource` (or `UpdateResFile`) procedures; then call `HSetState` when you're finished. (See Listing 1-2 on page 1-21 for an example of this technique.) However, most applications do not make resources purgeable and therefore don't need to take this precaution.

Here's an example of a situation in which an application might need to write a resource. As previously described, the SurfWriter application always saves the last position of a document window when the user saves the document, storing this information in a resource that it has defined for this purpose. SurfWriter defines a resource with resource type `rWinState` and resource ID `kLastWinStateID` to store the window state (its position and state, that is, either the user or the standard state). SurfWriter's window state resource has this format, defined by a record of type `MyWindowState`:

```
TYPE MyWindowState =
   RECORD
      userStateRect: Rect;    {user state rectangle}
      zoomState:     Boolean; {window state: TRUE = standard; }
                              {              FALSE = user}
   END;

MyWindowStatePtr = ^MyWindowState;
MyWindowStateHnd = ^MyWindowStatePtr;
```

Listing 1-11 shows SurfWriter's application-defined routine for saving the last position of a window in a window state resource in a document's resource fork.

**Listing 1-11** Saving a resource to a resource fork

```
PROCEDURE MySaveWindowPosition (myWindow: WindowPtr;
                                myResFileRefNum: Integer);
VAR
   lastWindowState:  MyWindowState;
   myStateHandle:    MyWindowStateHnd;
   curResRefNum:     Integer;
BEGIN
   {set user state provisionally and determine whether window is zoomed}
   lastWindowState.userStateRect := WindowPeek(myWindow)^.contRgn^^.rgnBBox;
   lastWindowState.zoomState := EqualRect(lastWindowState.userStateRect,
                                           MyGetWindowStdState(myWindow));
   {if window is in standard state, then set the window's user state from }
   { the userStateRect field in the state data record}
   IF lastWindowState.zoomState THEN      {window was in standard state}
      lastWindowState.userStateRect := MyGetWindowUserState(myWindow);
   curResRefNum := CurResFile;   {save the refNum of current resource file}
   UseResFile(myResFileRefNum);  {set the current resource file}
   myStateHandle := MyWindowStateHnd(Get1Resource(rWinState,
                                                   kLastWinStateID));
   IF myStateHandle <> NIL THEN        {a state data resource already exists}
   BEGIN                               {update it}
      myStateHandle^^ := lastWindowState;
      ChangedResource(Handle(myStateHandle));
      IF ResError <> noErr THEN
         DoError;
   END
   ELSE                                {no state data has yet been saved}
   BEGIN                               {add state data resource}
      myStateHandle := MyWindowStateHnd(NewHandle(SizeOf(MyWindowState)));
      IF myStateHandle <> NIL THEN
      BEGIN
         myStateHandle^^ := lastWindowState;
         AddResource(Handle(myStateHandle), rWinState, kLastWinStateID,
                     'last window state');
      END;
   END;
   IF myStateHandle <> NIL THEN
   BEGIN
      UpdateResFile(myResFileRefNum);
      ReleaseResource(Handle(myStateHandle));
   END;
   UseResFile(curResRefNum);
END;
```

The `MySaveWindowPosition` procedure first sets the `userStateRect` field of the window state record to the bounds of the current content region of the window. It also sets the `zoomState` field of the record to a Boolean value that indicates whether the window is currently in the user state or standard state. If the window is in the standard state, the procedure sets the `userStateRect` field of the window state record to the user state of the window. (SurfWriter always saves the user state and the last state of the window. When it opens a document, it sets the user state to its previous state, verifies that this position is still valid, then calculates the window's standard state.)

The `MySaveWindowPosition` procedure then saves the file reference number of the current resource file and sets the current resource file to the document displayed in the current window. The procedure then uses the `Get1Resource` function to determine whether the resource file of the document already contains a window state resource. If so, the procedure changes the resource data, then calls `ChangedResource` to set the `resChanged` attribute of the resource's entry of the resource map in memory. If the resource doesn't yet exist, the procedure simply adds the new resource using the `AddResource` procedure.

Note that when a Resource Manager routine returns a handle to a resource, it returns the resource using the `Handle` data type. You usually define a data type (such as `MyWindowState`) to access the resource's data. If you also define a handle to your defined data type (such as `MyWindowStateHnd`), you need to coerce the returned handle to the appropriate type, as shown in this line from Listing 1-11:

```
myStateHandle := MyWindowStateHnd(Get1Resource(rWinState, kLastWinStateID));
```

If you use this method, you also need to coerce your defined handle back to a handle of type `Handle` when you use other Resource Manager routines, as shown in this line from Listing 1-11:

```
AddResource(Handle(myStateHandle), rWinState, kLastWinStateID,
            'last window state');
```

After `MySaveWindowPosition` changes or adds the resource (affecting only the resource map and resource data in memory), the `MySaveWindowPosition` procedure makes the change permanent by calling `UpdateResFile` and specifying the file reference number of the resource fork to update on disk. The `UpdateResFile` procedure writes the entire resource map in memory to disk and updates the resource data of any resource whose `resChanged` attribute is set in the resource map in memory.

(If you want to update only the resource that was just changed or added, you can use `WriteResource` instead of `UpdateResFile`.)

**Note**
Listing 1-11 assumes the window state resource is not purgeable. If it were, `MySaveWindowPosition` would need to call `HGetState` and `HNoPurge` before changing the resource. ◆

When done with the resource, `MySaveWindowPosition` uses `ReleaseResource`, which releases the memory allocated to the resource's data (and at the same time sets the master pointer of the resource's handle in the resource map in memory to `NIL`). Then `MySaveWindowPosition` restores the current resource file to its previous setting.

## Working With Partial Resources

Some resources, such as the `'snd '` and `'sfnt'` resources, can be quite large—sometimes too large to fit in the available memory. The `ReadPartialResource` and `WritePartialResource` procedures, which are available in System 7 and later versions of system software, allow you to read a portion of the resource into memory or alter a section of the resource while it is still on disk. You can also use the `SetResourceSize` procedure to enlarge or reduce the size of a resource on disk. When you use `ReadPartialResource` and `WritePartialResource`, you specify how far into the resource you want to begin reading or writing and how many bytes you actually want to read or write at that spot, so you must be sure of the location of the data.

▲ **WARNING**
Be aware that having a copy of a resource in memory when you are using the partial resource routines may cause problems. For example, if you read the resource into memory using `GetResource`, modify the resource in memory, and then access the resource on disk using either the `ReadPartialResource` or `WritePartialResource` procedure, note that these procedures work with the data in the buffer you specify, not the data referenced through the resource's handle. ▲

To read or write any part of a resource, call the `SetResLoad` procedure specifying `FALSE` for its `load` parameter, then use the `GetResource` function to get an empty handle (that is, a handle whose master pointer is set to `NIL`) to the resource. (Because of the call to the `SetResLoad` procedure, the `GetResource` function does not load the entire resource into memory.) Then call `SetResLoad` specifying `TRUE` for its `load` parameter and use the partial resource routines to access portions of the resource.

Listing 1-12 illustrates one way to deal with partial resources. The application-defined procedure `MyReadAPartial` begins by calling `SetResLoad` (with the `load` parameter set to `FALSE`) to ensure that the Resource Manager will not attempt to read the entire resource into memory in the subsequent call to `GetResource`. After calling `GetResource` and checking for errors, `MyReadAPartial` calls `SetResLoad` (with the `load` parameter set to `TRUE`) to restore normal loading of resource data into memory. The procedure then calls `ReadPartialResource`, specifying as parameters the handle returned by `GetResource`, an offset to the beginning of the resource subsection to be read, a buffer into which to read the subsection, and the length of the subsection. The `ReadPartialResource` procedure reads the specified partial resource into the specified buffer.

**Listing 1-12** Using partial resource routines

```
PROCEDURE MyReadAPartial(myRsrcType: ResType; myRsrcID: Integer;
                         start: LongInt; count: LongInt;
                         VAR putItHere: Ptr);
VAR
   myResHdl:         Handle;
   myErr:            OSErr;
BEGIN
   SetResLoad(FALSE);      {don't load resource}
   myResHdl := GetResource(myRsrcType, myRsrcID);
   myErr := ResError;
   SetResLoad(TRUE);       {reset to always load}
   IF myErr = noErr THEN
   BEGIN
      ReadPartialResource(myResHdl, start, putItHere, count);
      myErr := ResError;
      {check and report error}
      IF myErr <> noErr THEN DoError(myErr);
   END
   ELSE {handle error from GetResource}
      DoError(myErr);
END;
```