

# Resource Manager

---

## Contents

Introduction to Resources	1-3
The Data Fork and the Resource Fork	1-4
Resource Types and Resource IDs	1-6
The Resource Map	1-8
Search Path for Resources	1-10
About the Resource Manager	1-12
Using the Resource Manager	1-13
Creating a Resource	1-15
Getting a Resource	1-18
Releasing and Detaching Resources	1-22
Opening a Resource Fork	1-24
Opening an Application's Resource Fork	1-24
Creating and Opening a Resource Fork	1-25
Specifying the Current Resource File	1-28
Reading and Manipulating Resources	1-30
Writing Resources	1-36
Working With Partial Resources	1-40
Resource Manager Reference	1-42
Data Structure, Resource Types, and Resource IDs	1-42
The Resource Type	1-42
Resource IDs	1-46
Resource IDs of Owned Resources	1-47
Resource Names	1-49
Resource Manager Routines	1-49
Initializing the Resource Manager	1-50
Checking for Errors	1-51
Creating an Empty Resource Fork	1-53
Opening Resource Forks	1-58
Getting and Setting the Current Resource File	1-68
Reading Resources Into Memory	1-71

## CHAPTER 1

Getting and Setting Resource Information	1-81
Modifying Resources	1-87
Writing to Resource Forks	1-92
Getting a Unique Resource ID	1-95
Counting and Listing Resource Types	1-97
Getting Resource Sizes	1-104
Disposing of Resources	1-106
Closing Resource Forks	1-110
Reading and Writing Partial Resources	1-111
Getting and Setting Resource Fork Attributes	1-116
Accessing Resource Entries in a Resource Map	1-119
Resource File Format	1-121
Resources in the System File	1-126
User Information Resources	1-127
Packages	1-128
Function Key Resources	1-129
Standard Icons	1-129
ROM Resources	1-134
Inserting the ROM Resource Map	1-134
Overriding ROM Resources	1-135
Summary of the Resource Manager	1-137
Pascal Summary	1-137
Constants	1-137
Data Type	1-139
Routines	1-139
C Summary	1-142
Constants	1-142
Data Type	1-143
Routines	1-144
Assembly-Language Summary	1-147
Trap Macros	1-147
Global Variables	1-147
Result Codes	1-148

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.

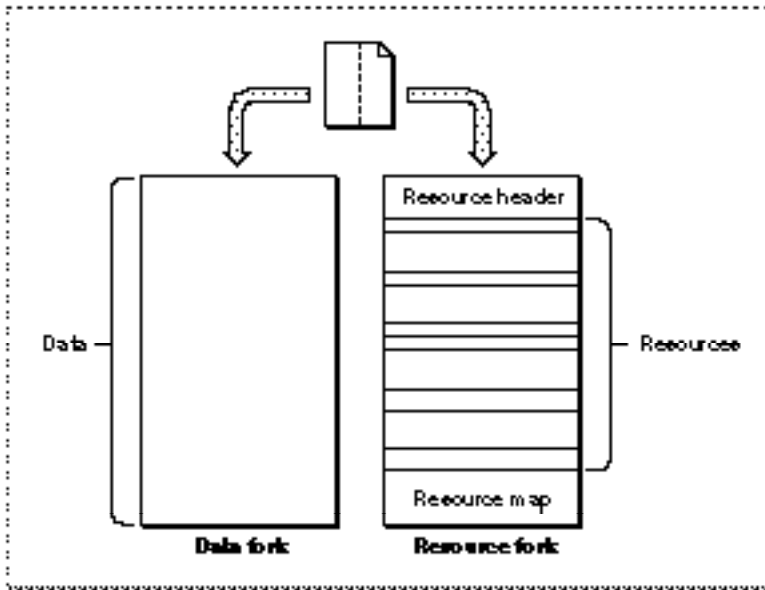
*Inside Macintosh: Macintosh Toolbox Essentials* describes how other managers, such as the Menu Manager, Window Manager, Dialog Manager, and Control Manager, use the Resource Manager to read resources for you. For example, you can use the Menu Manager, Window Manager, Dialog Manager, and Control Manager 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.

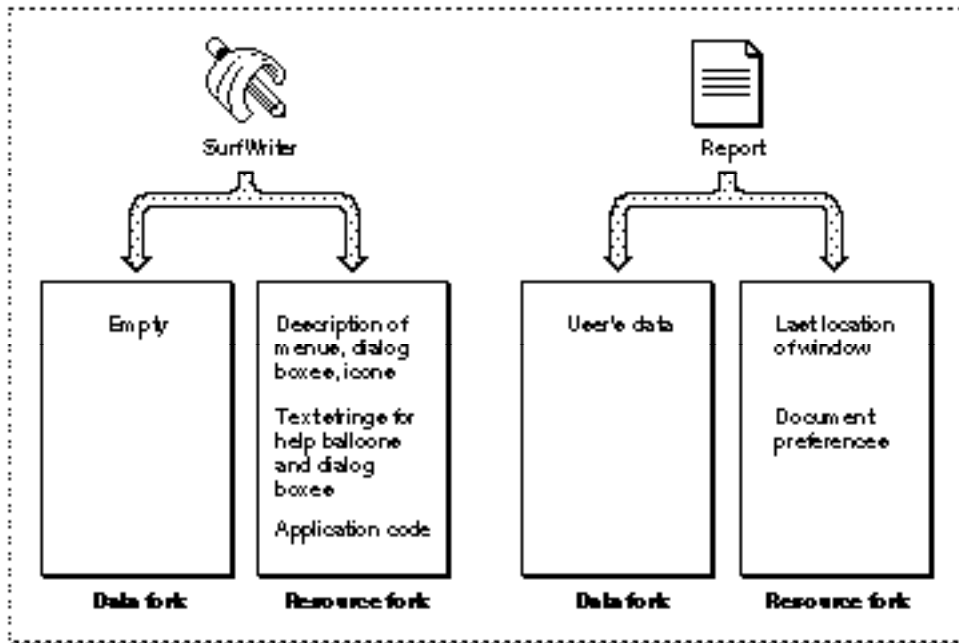
Whether you store data in the data fork or the resource fork of a document file depends largely on whether you can structure that data in a useful manner as a resource. For example, it's often convenient to store document-specific settings, such as the document's previous window size and location, as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

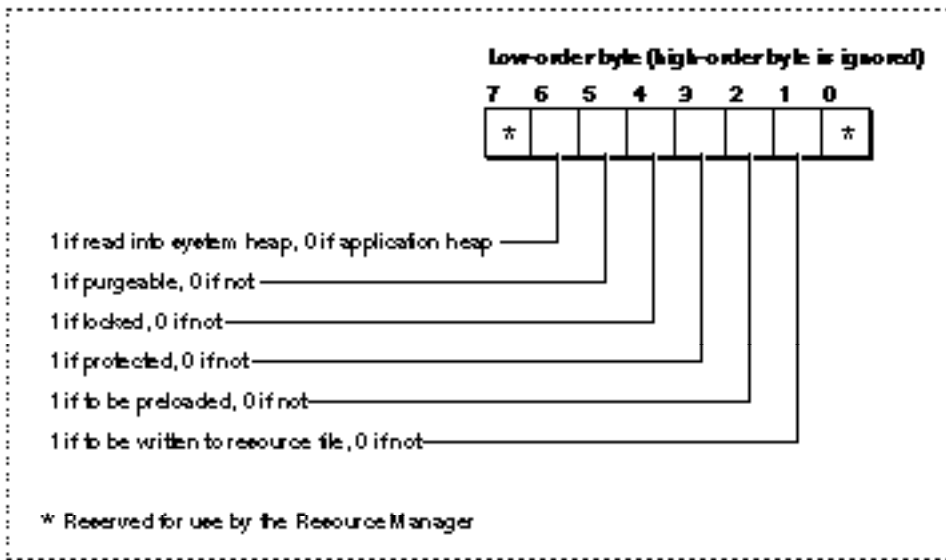
The **resource map** in the resource fork of a file contains entries for each resource in the resource fork. Each entry lists the resource’s resource type, resource ID, name, attributes, and location. When the Resource Manager opens the resource fork of a file, it reads the resource map into memory.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



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.

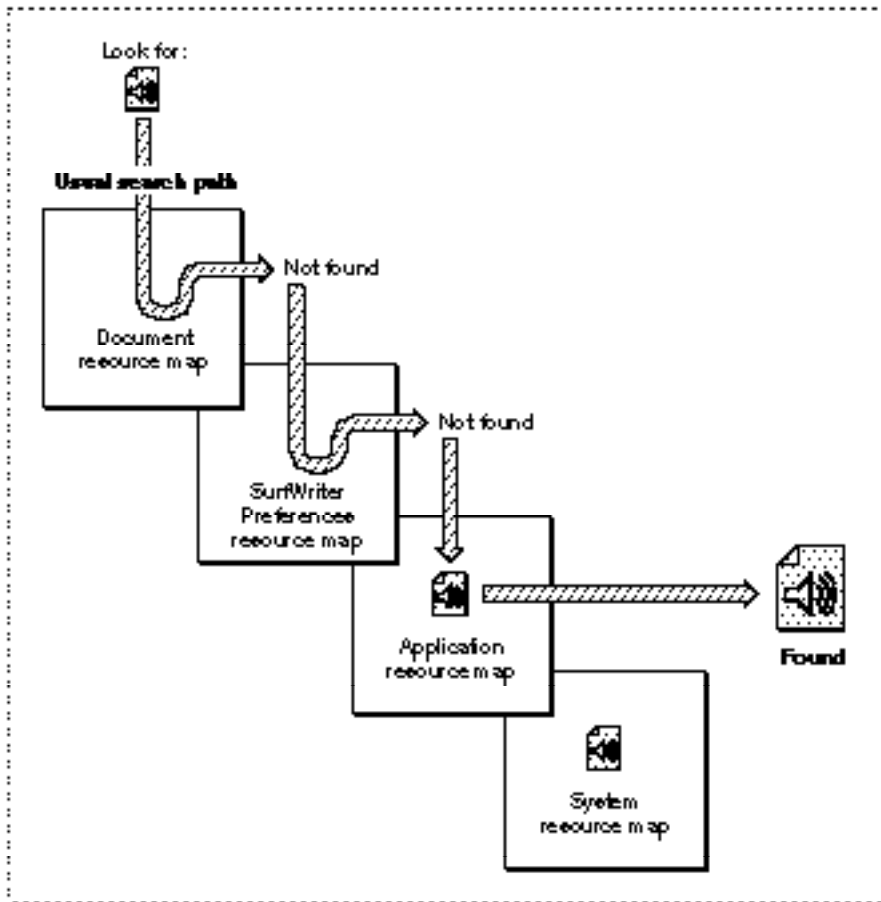
This default search order allows your application to use resources defined in the System file, to override resources defined in the System file, to share a single resource among several files by storing it in your application's resource fork, and to override application-defined resources with document-specific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

## Resource Manager

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

<b>Resource fork</b>	<b>Resources contained in resource fork</b>
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*.

## CHAPTER 1

### Resource Manager

```
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

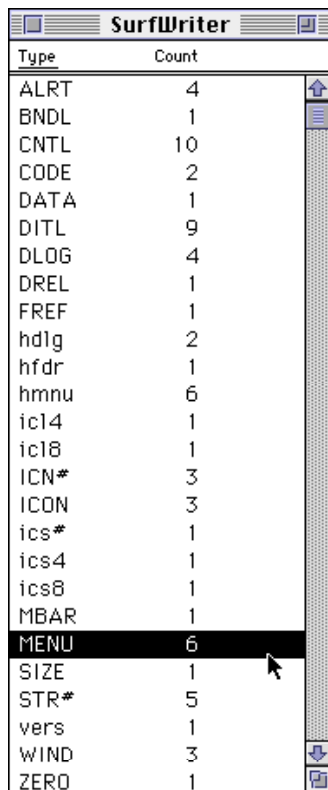
- get a handle to a resource and modify a purgeable resource safely
- release and detach resources
- create and open a resource fork
- set the current resource file (the file whose resource fork the Resource Manager searches first)
- read and manipulate resources
- write resources
- read and write partial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



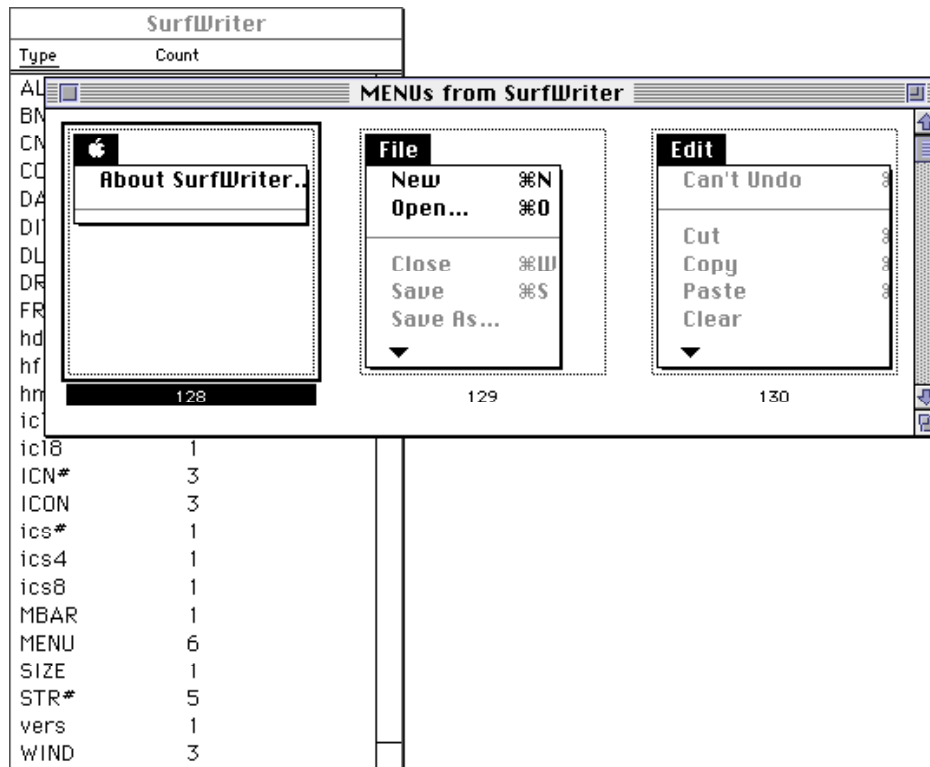
Type	Count
ALRT	4
BNDL	1
CNTL	10
CODE	2
DATA	1
DITL	9
DLOG	4
DREL	1
FREF	1
hdlg	2
hldr	1
hmnv	6
ic14	1
ic18	1
ICN#	3
ICON	3
ics#	1
ics4	1
ics8	1
MBAR	1
<b>MENU</b>	<b>6</b>
SIZE	1
STR#	5
vers	1
WIND	3
ZERO	1

## CHAPTER 1

### Resource Manager

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





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.)

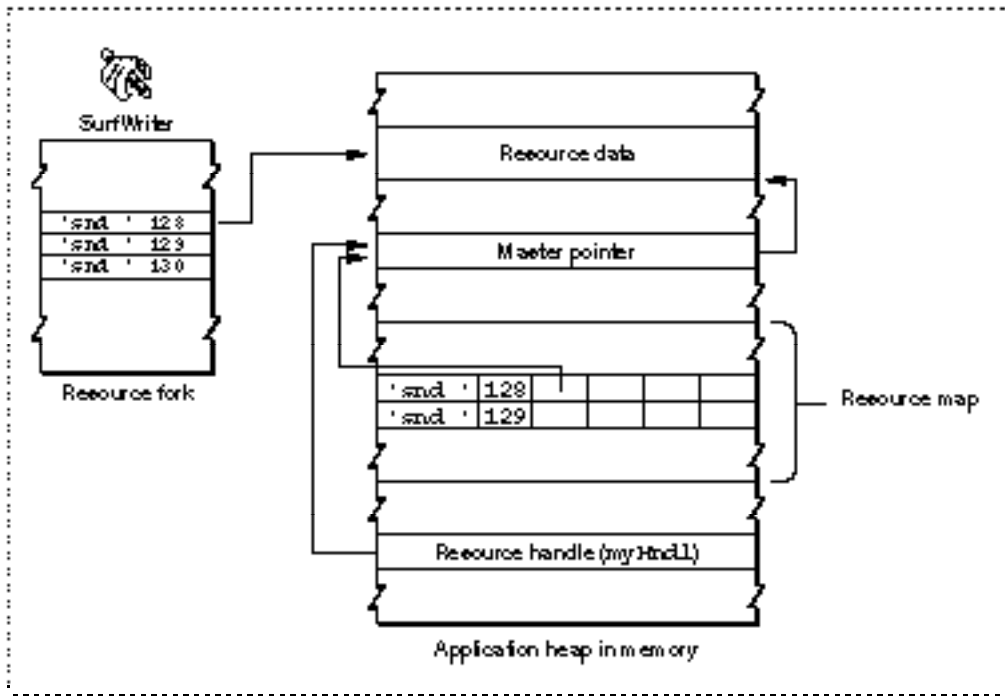
For example, this code uses `GetResource` to get a handle to an `'snd '` resource with resource ID 128.

```
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.

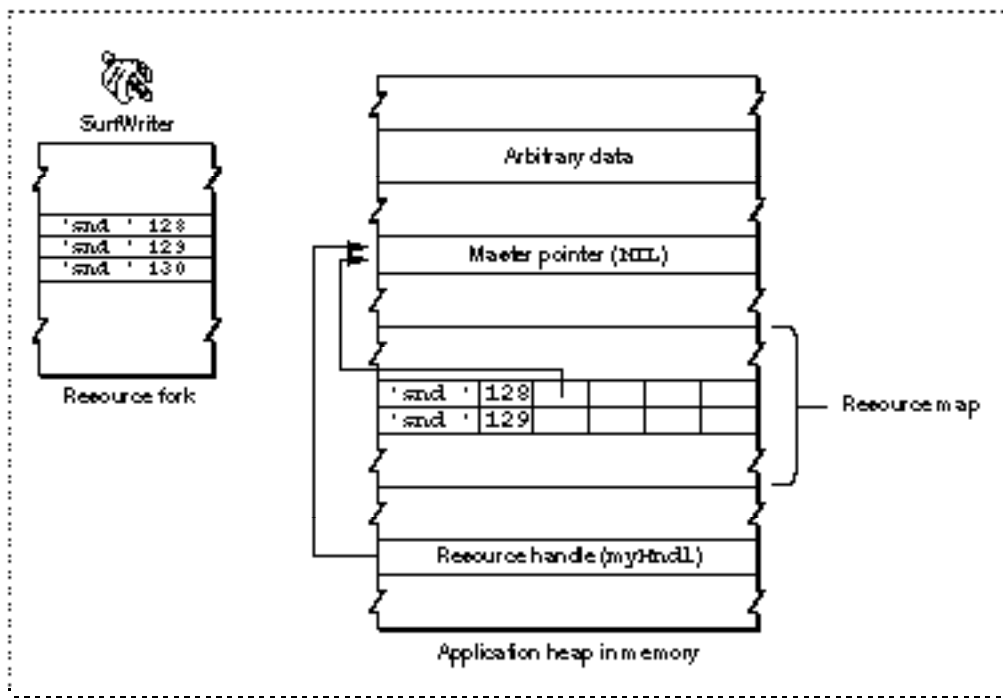
Figure 1-7 Getting a handle to a resource



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).

---

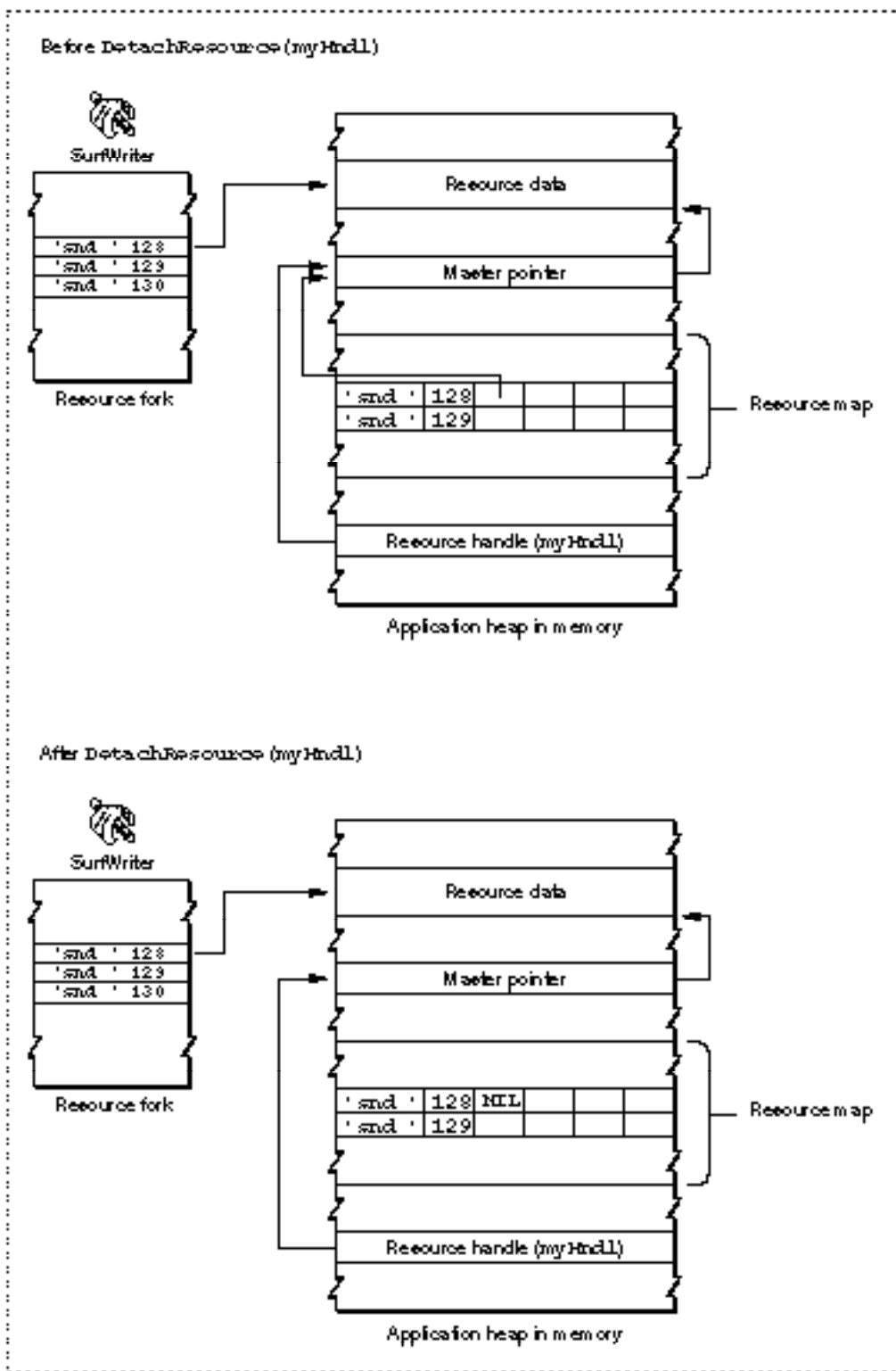
### Listing 1-3 Releasing a resource

```
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;

```

## Resource Manager

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 `rWindowState` and resource ID `kLastWindowStateID` 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;

```

## CHAPTER 1

### Resource Manager

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;
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));
```

## Resource Manager

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;
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;

```

## CHAPTER 1

### Resource Manager

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;
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;
```

## Resource Manager Reference

---

This section begins by describing the data type, standard resource types, and ranges of resource IDs used for various kinds of resources. “Resource Manager Routines” beginning on page 1-49 describes the routines provided by the Resource Manager for manipulating resources.

“Resource File Format” beginning on page 1-121 describes the format of a resource fork. “Resources in the System File” beginning on page 1-126 describes System file resources such as packages and icons. “ROM Resources” beginning on page 1-134 describes how to access ROM resources directly and how to override them.

### Data Structure, Resource Types, and Resource IDs

---

This section describes the data type for the resource type, lists the standard resource types, and describes the ranges of resource IDs available to your application for different kinds of resources. The Resource Manager and your application use a resource type and a resource ID to identify a specific resource.

### The Resource Type

---

The Resource Manager uses the resource type along with the resource ID to identify a resource uniquely. A resource type is defined by the `ResType` data type.

```
TYPE ResType = PACKED ARRAY[1..4] OF Char;
```

A resource type can be any sequence of four alphanumeric characters, including the space character. You can define your own resource types, but they must consist of all uppercase letters and must not conflict with any of the standard resource types.

#### **IMPORTANT**

When identifying resource types, the Resource Manager distinguishes between uppercase letters and their lowercase counterparts. In addition, Apple reserves for its own use all resource types that consist of all lowercase letters, all spaces, or all international characters (characters greater than \$7F). ▲

Table 1-2 lists the standard resource types.

**Table 1-2** Standard resource types

<b>Resource type</b>	<b>Description</b>
'ADBS'	Apple Desktop Bus service routine
'ALRT'	Alert box
'BNDL'	Bundle
'CDEF'	Control definition function
'CDEV'	Control device function for a control panel
'CNTL'	Control
'CODE'	Application code segment
'CURS'	Cursor
'DITL'	Item list in a dialog or alert box
'DLOG'	Dialog box
'DRVVR'	Desk accessory or other device driver
'FKEY'	Command-Shift-number combination
'FOND'	Font family record
'FONT'	Bitmapped font
'FREF'	File reference
'ICN#'	Large (32-by-32 pixel) black-and-white icon, with mask
'ICON'	Large (32-by-32 pixel) black-and-white icon, without mask
'INIT'	System extension
'KCAP'	Physical keyboard description (used by Key Caps desk accessory)
'KCHR'	Keyboard layout (software); maps virtual key codes to character codes
'LDEF'	List definition procedure
'MBAR'	Menu bar
'MDEF'	Menu definition procedure
'MENU'	Menu
'NFNT'	Bitmapped font
'PACK'	Package
'PAT'	Pattern
'PAT#'	Pattern list

*continued*

**Table 1-2** Standard resource types (continued)

<b>Resource type</b>	<b>Description</b>
'PICT'	QuickDraw picture
'POST'	PostScript <sup>®</sup> resource
'PREC'	Print record
'SICN'	Small (16-by-16 pixel) icon (mask optional)
'SIZE'	Size of application's partition and other information
'STR '	String
'STR#'	String list
'WDEF'	Window definition function
'WIND'	Window
'actb'	Alert color table
'alis'	Alias record
'card'	Video card name
'cctb'	Control color table
'cicn'	Color icon
'clut'	Color look-up table
'crsr'	Color cursor
'dctb'	Dialog color table
'ddev'	Database extension
'eadr'	Ethernet hardware address
'fctb'	Font color table
'hdlg'	Help for dialog box or alert box items
'hfdr'	Help for application icons
'hmnv'	Help for application menus
'hovr'	Help that overrides Finder help
'hrct'	Help for areas in windows
'hwin'	Association of 'hrct' and 'hdlg' resources to specific windows
'icl4'	Large (32-by-32 pixel) color icon with 4 bits of color data per pixel
'icl8'	Large (32-by-32 pixel) color icon with 8 bits of color data per pixel
'ics#'	Small (16-by-16 pixel) black-and-white icon, with mask
'ics4'	Small (16-by-16 pixel) color icon with 4 bits of color data per pixel
'ics8'	Small (16-by-16 pixel) color icon with 8 bits of color data per pixel

**Table 1-2** Standard resource types (continued)

<b>Resource type</b>	<b>Description</b>
'ictb'	Item color table
'itl0'	Date and time formats
'itl1'	Names of days and months
'itl2'	Text Utilities sort hooks
'itl4'	Localizable tables and code
'itlk'	Remappings of certain key combinations before the <code>KeyTrans</code> function is called for the corresponding 'KCHR' resource
'kcs#'	List of small black-and-white icons, with mask, for a corresponding 'KCHR' resource
'kcs4'	Small (16-by-16 pixel) color icon with 4 bits of color data per pixel for a corresponding 'KCHR' resource
'kcs8'	Small (16-by-16 pixel) color icon with 8 bits of color data per pixel for a corresponding 'KCHR' resource
'mctb'	Menu color information table
'mntr'	Monitors extension code resource
'movv'	QuickTime movie
'pltt'	Color palette
'ppat'	Pixel pattern
'qdef'	Query definition function
'qrsc'	Query resource
'sect'	Section record
'sfnt'	Outline font
'snd'	Sound
'snth'	Synthesizer
'styl'	TextEdit style record
'sysz'	System heap space required by a system extension
'vers'	Version number
'wctb'	Window color table
'wstr'	String (uses word for length byte)

Table 1-3 lists resource types that are reserved for use by system software. These resource types consist entirely of uppercase letters or combinations of uppercase and lowercase letters and the number sign (#). Other resource types specific to system software that consist entirely of lowercase letters or other characters are not included in Table 1-3. This list is provided for your information; you should not use these resource types in your application.

**Table 1-3** Resource types reserved for use by system software

Resource type	Description
' CACH '	RAM cache code
' DSAT '	System startup alert table
' FCMT '	“Get Info” comments
' FMTR '	3.5-inch disk formatting code
' FOBJ '	Folder information for an MFS volume
' FRSV '	IDs of system fonts
' INTL '	International resource (obsolete)
' KMAP '	Keyboard mapping (hardware); maps raw key codes to virtual key codes
' KSWP '	Defines special key combinations for Script Manager operations
' MBDF '	Default menu definition function
' MMAP '	Mouse-tracking code
' NBPC '	AppleTalk bundle
' PDEF '	Printing code
' PTCH '	ROM patch code
' ROv# '	List of ROM resources to override
' ROvr '	Code for overriding ROM resources
' SERD '	RAM Serial Driver

## Resource IDs

A resource is identified by its resource type and resource ID (or, optionally, its resource type and resource name). The IDs for resources used by the system software and those used by applications are assigned from separate ranges. By using these ranges correctly, you can avoid resource ID conflicts.

In general, system resources use IDs in the range -32767 through 127, and application resources must use IDs that fall between 128 and 32767. The IDs for some categories of resources, such as definition procedures and font families, fall in different ranges or in ranges that are broken down for more specific purposes. This list shows the resource ID ranges used for most resources.

Range	Description
-32768 through -16385	Reserved; do not use. Any application resource whose ID is in this range will not work properly in current versions of system software.
-16384 through -4065	Used for system resources owned by other system resources.
-4064 through -4033	Reserved for use by control panels. (See the chapter “Control Panels” in this book.)
-4032 through -1	Used for system resources owned by other system resources. The exception is the 'SIZE' resource, whose ID can be -1, 0 (preferred size), or 1 (minimum size).
0 through 127	Used for system resources and any definition procedures in the system software. Applications should not use these resource IDs.
128 through 32767	Available for your use. Your application's definition procedures should use IDs in the range 128 through 4095, although other resources may use these IDs as well. Font families for individual script systems have additional restrictions defined in the appendix on international resources in <i>Inside Macintosh: Text</i> .

For a general discussion of font family resource IDs, see *Inside Macintosh: Text*.

The ID range of definition procedures (which are usually contained in resources such as the 'WDEF' or 'CDEF' resources) is limited to 12 bits (0 through 4095). The system software's own definition procedures, which are located in the System file, have resource IDs from 0 through 127. The IDs of your definition procedures should be in the range 128 through 4095.

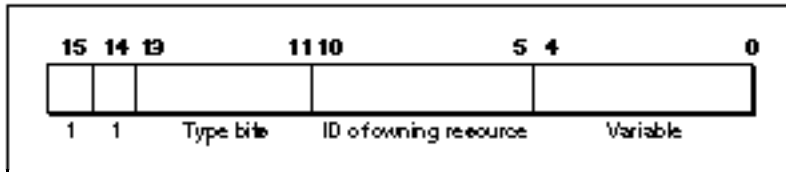
## Resource IDs of Owned Resources

---

Certain types of resources used by system software may have resources of their own in the same resource fork; the “owning” resource consists of code that reads the “owned” resource into memory. For example, a desk accessory might have its own pattern and string resources. This section describes the numbering convention used for owned resources. This information can be useful if you are writing a desk accessory or other driver or special types of definition functions for windows, controls, or menus.

You should use the numbering convention described in this section to associate owned resources with the resources to which they belong. This allows resource-copying programs (such as installers) to recognize which additional resources need to be copied along with an owning resource. Figure 1-10 illustrates the ID of an owned resource.

**Figure 1-10** Resource ID of an owned resource



Bits 14 and 15 are always 1. Bits 11 through 13 specify the type of the owning resource, as follows:

Type bits	Type
000	'DRVR'
001	'WDEF'
010	'MDEF'
011	'CDEF'
100	'PDEF'
101	'PACK'
110	Reserved for future use
111	Reserved for future use

Bits 5 through 10 contain the resource ID of the owning resource (limited to 0 through 63). Bits 0 through 4 contain any desired value (0 through 31).

Some types of resources can't be owned because their IDs don't conform to this convention. For example, a resource of type 'WDEF' can own other resources but cannot itself be owned, because its resource ID can't be more than 12 bits long (see the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*). The chapters describing individual resources provide detailed information about such restrictions.

An owned resource may itself contain the ID of a resource associated with it. For example, a dialog ('DLOG') resource owned by a desk accessory contains the resource ID of its item list. Although the item list is associated with the dialog resource, it's actually owned (indirectly) by the desk accessory. The resource ID of the item list should conform to the same special convention as the ID of the dialog resource. For example, if the resource ID of the desk accessory is 17, the IDs of both the dialog resource and the item list should contain the value 17 in bits 5 through 10.



A program that copies resources may need to change the resource ID of a resource so as not to duplicate an existing resource ID. Bits 5 through 10 of resources owned, directly or indirectly, by the copied resource should also be changed when those resources are copied. In the example just discussed, if the desk accessory must be given a new ID, bits 5 through 10 of both the dialog resource and the item list resource should also change.

▲ **WARNING**

When a resource-copying program changes the ID of an owned resource, it should also change the ID where it appears in other resources (such as an item list's ID contained in a dialog box resource). ▲

## Resource Names

---

You can use a resource name instead of a resource ID to identify a resource of a given type. Like a resource ID, a resource name should be unique within each type. If you assign the same resource name to two resources of the same type, the second assignment of the name overrides the first, thereby making the first resource inaccessible by name. When comparing resource names, the Resource Manager ignores case (but does not ignore diacritical marks).

## Resource Manager Routines

---

This section describes the routines provided by the Resource Manager. You can use these routines to create, open, and close resource forks; get and set the current resource file; read resources into memory; get and set resource information; modify resources; write to resource forks on disk; get a unique resource ID; count and list resource types; get resource sizes; dispose of resources; read and write partial resources; get and set resource fork attributes; and access resource entries in the resource map.

The `FSpCreateResFile` procedure and the `FSpOpenResFile` function use a file system specification (`FSSpec`) record. These routines are available only in System 7 or later. Use the `Gestalt` function to determine if these routines are available. If they're not available, you can call the equivalent File Manager HFS routines, the `HCreateResFile` procedure and the `HOpenResFile` function.

The Resource Manager provides a means for reporting errors specifically related to resources. After calling a Resource Manager routine, you can call the `ResError` function to determine whether any error occurred. The `ResError` function returns an integer value identifying any error reported by the Resource Manager routine that was executed last. The values listed in the `ResError` description signify only those errors dealing specifically with resources. The `ResError` function can also return values corresponding to Operating System result codes. The description for each Resource Manager routine includes the errors `ResError` may report for that routine under the subheading "Result Codes"; this list includes both the integer result codes for the Resource Manager routine as well as common Operating System result codes.

## Initializing the Resource Manager

---

Unlike other Toolbox managers, the Resource Manager does not need to be explicitly initialized. System software automatically calls the Resource Manager's two initialization routines, the `InitResources` function and the `RsrcZoneInit` procedure—the former when the system starts up, and the latter when the system starts up and when the Process Manager starts up. You should not call either of these routines directly.

## InitResources

---

When the system starts up, it automatically calls the `InitResources` function. This routine is for system use only, and your application should not call it at any time.

```
FUNCTION InitResources: Integer;
```

### DESCRIPTION

The `InitResources` function initializes the Resource Manager. `InitResources` 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. The `InitResources` function returns an integer, which is the file reference number for the System file's resource fork.

Your application does not need to know the file reference number for the System file's resource fork, because every Resource Manager routine with a file reference number parameter also accepts 0 to mean the System file's resource fork.

### ASSEMBLY-LANGUAGE INFORMATION

The `InitResources` function sets up three global variables: `SysResName`, `SysMap`, and `SysMapHndl`. These contain, respectively, the name of the System file's resource fork, the file reference number for the resource fork, and a handle to the System file's resource map.

## RsrcZoneInit

---

System software automatically calls the `RsrcZoneInit` procedure when system software starts up and when the Process Manager starts up. Your application should not call this routine directly.

```
PROCEDURE RsrcZoneInit;
```

**DESCRIPTION**

System software automatically calls the `RsrcZoneInit` procedure at system startup when extensions are loaded, because each extension has its own application heap. System software calls `RsrcZoneInit` once again when the Process Manager starts up. After that, the procedure is not called again.

**Checking for Errors**

---

You can use the `ResError` function in your application to retrieve errors that may result from calling Resource Manager routines. You also can use `ResError` to check for an error after application startup (system software opens the resource fork of your application during application startup).

**ResError**

---

After calling a Resource Manager routine, you can use the `ResError` function to determine whether an error occurred and, if so, what it was.

```
FUNCTION ResError: Integer;
```

**DESCRIPTION**

The `ResError` function reads the value stored in the system global variable `ResErr` and returns an integer result code identifying errors, if any, that occurred. If no error occurred, `ResError` returns `noErr`. If an error occurs at the Resource Manager level, `ResError` returns one of the integer result codes listed in this section. If an error occurs at the Operating System level, `ResError` returns an Operating System result code, such as the Memory Manager error `memFullErr` or the File Manager error `ioErr`.

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`.

**IMPORTANT**

In certain cases, the `ResError` function returns `noErr` even though a Resource Manager routine was unable to perform the requested operation. See the individual routine descriptions for details about the circumstances under which this happens. ▲

## CHAPTER 1

### Resource Manager

Only those result codes dealing specifically with resources are listed in this section. See the description of each Resource Manager routine for a list of errors specific to that routine and that the `ResError` function returns.

#### ASSEMBLY-LANGUAGE INFORMATION

The global variable `ResErr` stores the current value of `ResError`, that is, the result code of the most recently performed Resource Manager operation. In addition, you can specify an application-defined procedure to be called whenever an error occurs. To do this, store the address of the procedure in the global variable `ResErrProc`. The value of the `ResErrProc` global variable is usually 0. Before returning a result code other than `noErr`, the `ResError` function puts that result in register D0 and calls the procedure identified by the `ResErrProc` global variable.

If you use `ResErrProc` to detect resource errors, you will get unexpected calls to your application-defined procedure if you call `GetMenu`. The Menu Manager routine `GetMenu` makes a call to `GetResInfo`, requesting resource information about 'MDEF' 0. Unfortunately, because `ROMMapInsert` is set to `FALSE`, this call fails, setting `ResErr` to `-192` (`resNotFound`). This, in turn, causes a call to your application-defined procedure, even though the `GetMenu` routine has worked correctly.

To avoid this problem, follow these steps when you call `GetMenu` if you are using `ResErrProc`:

1. Save the address of your application-defined procedure.
2. Clear `ResErrProc`.
3. Call `GetResource` for the menu resource you want to get.
4. Check whether `GetResource` returns a handle whose value is `NIL`; if so, process the error in whatever way is appropriate for your application.
5. Call `GetMenu`.
6. When you are finished calling `GetMenu`, restore the previous value of `ResErrProc`.

#### RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>resFNotFound</code>	-193	Resource file not found
<code>addResFailed</code>	-194	AddResource procedure failed
<code>rmvResFailed</code>	-196	RemoveResource procedure failed
<code>resAttrErr</code>	-198	Attribute inconsistent with operation
<code>mapReadErr</code>	-199	Map inconsistent with operation

## Creating an Empty Resource Fork

---

You can use `FSpCreateResFile`, `HCreateResFile`, or `CreateResFile` when you want to create an empty resource fork—that is, a resource fork that contains no resource data but does include a resource map. Note that creating a resource fork does not automatically open it. To open a resource fork of a file created with one of these routines, use the corresponding routines `FSpOpenResFile`, `HOpenResFile`, or `OpenResFile`.

The `FSpCreateResFile` procedure is available only in System 7 and later versions of system software. If `FSpCreateResFile` is not available, you can use `HCreateResFile` or `CreateResFile` to create a resource fork. The `HCreateResFile` procedure allows you to specify a directory ID and a volume reference number, and is therefore preferred over `CreateResFile`. The `CreateResFile` procedure is an earlier version of `HCreateResFile` that is still supported but has more restricted capabilities.

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.

The File Manager assumes that the first block of a file's resource fork is part of the resource header and puts information there that it uses during scavenging—for example, after the user presses the Reset switch. For this reason, if you copy a resource file, the duplicate may not be exactly like the original.

## FSpCreateResFile

---

You can use the `FSpCreateResFile` procedure to create an empty resource fork using a file system specification (`FSSpec`) record.

```
PROCEDURE FSpCreateResFile (spec: FSSpec;
                           creator, fileType: OSType;
                           scriptTag: ScriptCode);
```

<code>spec</code>	A file system specification record that indicates the name and location of the file whose resource fork is to be created.
<code>creator</code>	The signature of the application creating the file.
<code>fileType</code>	The file type of the new file.
<code>scriptTag</code>	The script code of the script system in which the Finder and standard file dialog boxes display the file's name.

## DESCRIPTION

The `FSpCreateResFile` procedure creates an empty resource fork for a file with the specified type, creator, and script code in the location and with the name designated by the `spec` parameter. (An empty resource fork contains no resource data but does include a resource map.)

This procedure is available only in System 7 and later versions of system software. If `FSpCreateResFile` is not available to your application, you can use `HCreateResFile` or `CreateResFile`.

The `spec` parameter is a file system specification record, which is the standard format in System 7 and later versions for identifying a file or directory. The file system specification record for files and directories 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.

The `creator` parameter of `FSpCreateResFile` contains the signature of the application that creates the file. Whenever your application creates a document, it assigns a creator and a file type to that document. Typically your application sets its signature as the document’s creator.

The `fileType` parameter indicates the type of file. You can set the file type to a type especially defined for your application or one of the existing general types, such as `'TEXT'` for text (a stream of ASCII characters), or `'pref'` for a preferences file.

**Note**

The file type should be as descriptive of the file’s data format as possible. You should not use `'TEXT'` as a file type unless the document contains plain ASCII characters. ♦

The value of the `scriptTag` parameter is the script code of the script system in which the Finder and the Standard File Package dialog boxes display the name of the file. For example, to specify the Roman script system, specify the constant `smRoman` in the `scriptTag` parameter.

If the file specified by the file system specification record doesn’t already exist (that is, if it 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 file system specification record already exists and includes a resource fork with a resource map, `FSpCreateResFile` does nothing. If the data fork of the file specified by the file system specification 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.

If your application uses Standard File Package routines, note that the `StandardPutFile` procedure returns a standard file reply record that contains a file system specification record in the `sfFile` field.

Before you can work with the newly created file's resource fork, you must use the `FSpOpenResFile` function to open it.

#### SPECIAL CONSIDERATIONS

The `FSpCreateResFile` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

#### RESULT CODES

<code>noErr</code>	0	No error
<code>dirFulErr</code>	-33	Directory full
<code>dskFulErr</code>	-34	Disk full
<code>nsvErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>bdNamErr</code>	-37	Bad filename or volume name (perhaps zero length)
<code>tmfoErr</code>	-42	Too many files open
<code>wPrErr</code>	-44	Disk is write-protected
<code>fLckdErr</code>	-45	File is locked

#### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51. For information about using the `Gestalt` function to determine whether the `FSpCreateResFile` procedure is available, see "Using the Resource Manager," beginning on page 1-13. For a discussion of the use of the `FSpCreateResFile` procedure, see "Creating and Opening a Resource Fork" beginning on page 1-25. For a description of the `FSpOpenResFile` function, see page 1-58. For information about the `StandardPutFile` procedure and standard file reply records, see *Inside Macintosh: Files*. For more information on creators and file types, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.

#### ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `FSpCreateResFile` procedure are

Trap macro	Selector
<code>_HighLevelFSDispatch</code>	<code>\$000E</code>

**HCreateResFile**

---

If the `FSpCreateResFile` procedure is not available, you can use the `HCreateResFile` procedure to create an empty resource fork.

```
PROCEDURE HCreateResFile (vRefNum: Integer; dirID: LongInt;
                          fileName: Str255);
```

`vRefNum`     The volume reference number of the volume on which the file is located.  
`dirID`        The directory ID of the directory where the file is located.  
`fileName`     The name of the file whose resource fork is to be created.

**DESCRIPTION**

The `HCreateResFile` procedure creates a file with an empty resource fork in the directory specified by the `vRefNum` and `dirID` parameters. (An empty resource fork contains no resource data but does include a resource map.)

If no other file with the given name exists in the specified directory, `HCreateResFile` creates a resource file—that is, a resource fork, including a resource map. In this case the file has a zero-length data fork.

If a file with the specified name already exists and includes a resource fork with a resource map, `HCreateResFile` does nothing. If the data fork of the specified file already exists but the file has a zero-length resource fork, `HCreateResFile` creates an empty resource fork and resource map for the file.

Before you can work with the newly created file's resource fork, you must first use `HOpenResFile` or a related function to open it.

**SPECIAL CONSIDERATIONS**

The `HCreateResFile` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>dirFulErr</code>	-33	Directory full
<code>dskFulErr</code>	-34	Disk full
<code>nsvErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>bdNamErr</code>	-37	Bad filename or volume name (perhaps zero length)
<code>tmfoErr</code>	-42	Too many files open
<code>wPrErr</code>	-44	Disk is write-protected
<code>fLckdErr</code>	-45	File is locked



## SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.  
For a description of the `HOpenResFile` function, see page 1-62.

## CreateResFile

---

If the `FSpCreateResFile` procedure is not available, you can use the `CreateResFile` procedure to create an empty resource fork.

```
PROCEDURE CreateResFile (fileName: Str255);
```

`fileName`     The name of the file to be created.

## DESCRIPTION

The `CreateResFile` procedure creates a file with an empty resource fork in your application's default directory—that is, the directory in which your application is located.

If no other file with the given name exists in the default directory or any of the other directories searched by `PBOpenRF` (see the following section, “Special Considerations”), `CreateResFile` creates a resource file—that is, a resource fork, including a resource map. In this case the file has a zero-length data fork.

If a file with the specified name already exists and includes a resource fork with a resource map, `CreateResFile` does nothing. Call `ResError` to determine whether an error occurred. If the data fork of the specified file already exists but the file has a zero-length resource fork, `CreateResFile` creates an empty resource fork and resource map for the file.

Before you can work with the newly created file's resource fork, you must use `OpenResFile` or a related function to open it.

## SPECIAL CONSIDERATIONS

The `CreateResFile` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

The `CreateResFile` procedure first checks whether a file with the specified name exists. (If so, `ResError` returns the result code `dupFNErr`.) To perform this check, `CreateResFile` calls `PBOpenRF`, which looks first in the default directory for a file with the same name, then in the root directory of the boot volume (if the default directory is on the boot volume), and then in the System Folder (if one exists on the same volume as the default directory). It is thus impossible, for example, to use `CreateResFile` to create a file in the default directory if a file with the same name already exists in the System Folder. To avoid this problem, use `FSpCreateResFile` or `HCreateResFile` whenever possible.

## RESULT CODES

noErr	0	No error
dirFulErr	-33	Directory full
dskFulErr	-34	Disk full
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename or volume name (perhaps zero length)
tmfoErr	-42	Too many files open
wPrErr	-44	Disk is write-protected
fLckdErr	-45	File is locked
dupFNErr	-48	Another file with the same name exists in the default directory, the root directory of the boot volume, or the System Folder

## SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `OpenResFile` function, see page 1-66.

## Opening Resource Forks

---

To open a resource fork, the Resource Manager calls the appropriate File Manager routine and returns the file reference number that it gets from the File Manager. If the file reference number returned is greater than 0, you can use this number to refer to the resource fork in some other Resource Manager routines.

The `FSpOpenResFile`, `HOpenResFile`, `OpenRFPPerm`, and `OpenResFile` functions all open resource forks. Use the `FSpOpenResFile` function to open a resource fork using a file system specification (`FSSpec`) record. You can determine whether `FSpOpenResFile` is available by calling the `Gestalt` function with the `gestaltFSAttr` selector code.

If `FSpOpenResFile` is not available, you can use `HOpenResFile`, `OpenRFPPerm`, or `OpenResFile` to open a resource fork. The `HOpenResFile` function allows you to specify both a directory ID and a volume reference number, and is therefore preferred if `FSpOpenResFile` is not available. The `OpenRFPPerm` and `OpenResFile` functions are earlier versions of `HOpenResFile` that are still supported but are more restricted in their capabilities.

## FSpOpenResFile

---

You can use the `FSpOpenResFile` function to open a file's resource fork using a file system specification (`FSSpec`) record.

```
FUNCTION FSpOpenResFile (spec: FSSpec;
                        permission: SignedByte): Integer;
```

## Resource Manager

<code>spec</code>	A file system specification record specifying the name and location of the file whose resource fork is to be opened.
<code>permission</code>	A value that specifies a read/write permission combination.

## DESCRIPTION

The `FSpOpenResFile` function opens the resource fork of the file identified by the `spec` parameter. It also makes this file the current resource file.

This function is available only in System 7 and later versions of system software. If `FSpOpenResFile` is not available to your application, you can use `HOpenResFile`, `OpenRFPPerm`, or `OpenResFile` instead.

The `spec` parameter is a file system specification record, which is a standard format in System 7 and later versions for identifying a file or directory. The file system specification record for files and directories 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;

```

You can specify the access path permission for the resource fork by setting the `permission` parameter to one of these constants:

```

CONST
    fsCurPerm   = 0; {whatever is currently allowed}
    fsRdPerm    = 1; {read-only permission}
    fsWrPerm    = 2; {write permission}
    fsRdWrPerm  = 3; {exclusive read/write permission}
    fsRdWrShPerm= 4; {shared read/write permission}

```

Use `fsCurPerm` to request whatever permission is currently allowed. If write access is unavailable (because the file is locked or because the resource fork is already open with write access), then read permission is granted. Otherwise, read/write permission is granted.

Use `fsRdPerm` to request permission to read the file, and `fsWrPerm` to write to it. If write permission is granted, no other access paths are granted write permission. Because the File Manager doesn't support write-only access to a file, `fsWrPerm` is synonymous with `fsRdWrShPerm`.

Use `fsRdWrPerm` and `fsRdWrShPerm` to request exclusive or shared read/write permission, respectively. If your application is granted exclusive read/write permission, no users are granted permission to write to the file; other users may, however, be granted

## CHAPTER 1

### Resource Manager

permission to read the file. Shared read/write permission allows multiple access paths for writing and reading.

The Resource Manager reads the resource map from the specified file's resource fork into memory. It also reads into memory every resource in the resource fork whose `resPreload` attribute is set.

The `FSpOpenResFile` function returns a file reference number for the resource fork. You can use this reference number to refer to the resource fork in other Resource Manager routines.

If you attempt to use `FSpOpenResFile` to open a resource fork that is already open, `FSpOpenResFile` returns the existing file reference number or a new one, depending on the access permission for the existing access path. For example, your application receives a new file reference number after a successful request for read-only access to a file previously opened with write access, whereas it receives the same file reference number in response to a second request for write access to the same file. In this case, `FSpOpenResFile` doesn't make that file the current resource file.

If the `FSpOpenResFile` function fails to open the specified file's resource fork (for instance, because there's no file with the given file system specification record or because there are permission problems), it returns `-1` as the file reference number. Use the `ResError` function to determine what kind of error occurred.

You don't have to call `FSpOpenResFile` to open the System file's resource fork or an application file's resource fork. These resource forks are opened automatically when the system and the application start up, respectively. To get the file reference number for your application, call the `CurResFile` function after your application starts up and before you open any other resource forks.

The `FSpOpenResFile` function checks that the information in the resource map is internally consistent. If it isn't, `ResError` returns the result code `mapReadErr`.

To open a resource fork just for block-level operations, such as copying files without reading the resource map into memory, use the File Manager function `OpenRF`.

### SPECIAL CONSIDERATIONS

The `FSpOpenResFile` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

It's possible to create multiple, unique, read-only access paths to a resource fork using `FSpOpenResFile`; however, you should avoid doing so. If a resource fork is opened twice—once with read/write permission and once with read-only permission—two copies of the resource map exist in memory. If you change one of the resources in memory using one of the resource maps, the two resource maps become inconsistent and the file will appear damaged to the second resource map.

If you must use this technique for read-only access, call `FSpOpenResFile` immediately before your application reads information from the file and close the file immediately afterward. Otherwise, your application may get unexpected results.

If an application attempts to open a second access path with write access and the application is different from the one that originally opened the resource fork, `FSpOpenResFile` returns -1, and the `ResError` function returns the result code `opWrErr`.

If you want to open the resource fork for another application (or any resource fork other than your application's that includes 'CODE' resources), you must bracket your calls to `FSpOpenResFile` with calls to `SetResLoad` with the load parameter set to `FALSE` and then to `TRUE`. You must also avoid making intersegment calls while the other application's resource fork is open. If you don't do this, the Segment Loader Manager treats any preloaded 'CODE' resources as your code resources when you make an intersegment call that triggers a call to `LoadSeg` while the other application is first in the resource chain. In this case, your application can begin executing the other application's code, and severe problems will ensue. If you need to get 'CODE' resources from the other application's resource fork, you can still prevent the Segment Loader Manager problem by calling `UseResFile` with your application's file reference number to make your application the current resource file.

#### ASSEMBLY-LANGUAGE INFORMATION

A handle to the resource map for the most recently opened resource fork is stored in the global variable `TopMapHndl`. The trap macro and routine selector for the `FSpOpenResFile` are

Trap macro	Selector
<code>_HighLevelFSDispatch</code>	<code>\$0000</code>

#### RESULT CODES

<code>noErr</code>	0	No error
<code>nsvErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>bdNamErr</code>	-37	Bad filename or volume name (perhaps zero length)
<code>eofErr</code>	-39	End of file
<code>tmfoErr</code>	-42	Too many files open
<code>fnfErr</code>	-43	File not found
<code>opWrErr</code>	-49	File already open with write permission
<code>permErr</code>	-54	Permissions error (on file open)
<code>extFSErr</code>	-58	Volume belongs to an external file system
<code>memFullErr</code>	-108	Not enough room in heap zone
<code>dirNFErr</code>	-120	Directory not found
<code>mapReadErr</code>	-199	Map inconsistent with operation

#### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51. For information about using the `Gestalt` function to determine whether the `FSpOpenResFile` procedure is available, see "Using the Resource Manager" beginning on page 1-13. For an example of the use of `FSpOpenResFile` to open a resource fork, see Listing 1-7 on page 1-27.

For information about the `CurResFile` and `UseResFile` routines, see page 1-68 and page 1-69, respectively.

For more information about permission parameter constants or the `OpenRF` function, see *Inside Macintosh: Files*.

## HOpenResFile

---

If the `FSpOpenResFile` function is not available, you can use `HOpenResFile` to open a file's resource fork.

```
FUNCTION HOpenResFile (vRefNum: Integer; dirID: LongInt;
                      fileName: Str255;
                      permission: SignedByte): Integer;
```

`vRefNum`     The volume reference number of the volume on which the file is located.  
`dirID`        The directory ID of the directory where the file is located.  
`fileName`     The name of the file whose resource fork is to be opened.  
`permission`   A constant for one of the read/write permission combinations.

### DESCRIPTION

The `HOpenResFile` function opens the resource fork of the file with the name specified by the `fileName` parameter in the directory specified by the `vRefNum` and `dirID` parameters. It also makes this file the current resource file.

You can specify the access path permission for the resource fork by setting the `permission` parameter to one of these constants:

```
CONST
    fsCurPerm    = 0; {whatever is currently allowed}
    fsRdPerm     = 1; {read-only permission}
    fsWrPerm     = 2; {write permission}
    fsRdWrPerm  = 3; {exclusive read/write permission}
    fsRdWrShPerm = 4; {shared read/write permission}
```

See page 1-59 for information about specifying access path permission with `FSpOpenResFile`. The same information applies to `HOpenResFile`.

The Resource Manager reads the resource map from the resource fork of the specified file into memory. It also reads into memory every resource whose `resPreload` attribute is set.

The `HOpenResFile` function returns a file reference number for the file. You can use this file reference number to refer to the file in other Resource Manager routines. If the file's resource fork is already open, `HOpenResFile` returns the file reference number but does not make that file the current resource file.

If the `HOpenResFile` function fails to open the specified file's resource fork (because there's no file with the specified name or because there are permission problems), it returns `-1` as the file reference number. Use the `ResError` function to determine what kind of error occurred.

You don't have to call `HOpenResFile` to open the System file's resource fork or an application file's resource fork. These files are opened automatically when the system and the application start up, respectively. To get the file reference number for your application, call the `CurResFile` function after the application starts up and before you open the resource forks for any other files.

The `HOpenResFile` function checks that the information in the resource map is internally consistent. If it isn't, `ResError` returns the result code `mapReadErr`.

To open a resource fork just for block-level operations, such as copying files without reading the resource map into memory, use the File Manager function `OpenRF`.

#### SPECIAL CONSIDERATIONS

The `HOpenResFile` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

It's possible to create multiple, unique, read-only access paths to a resource fork using `HOpenResFile`; however, you should avoid doing so. See page 1-60 for discussion of this issue in relation to `FSpOpenResFile`. The `HOpenResFile` function works the same way.

Versions of system software before System 7 do not allow you to use `HOpenResFile` to open a second access path, with write access, to a resource fork. In this case, `HOpenResFile` returns the reference number already assigned to the file.

If you want to open the resource fork for another application (or any resource fork other than your application's that includes 'CODE' resources), you must bracket your calls to `HOpenResFile` with calls to `SetResLoad` with the load parameter set to `FALSE` and then to `TRUE`. You must also avoid making intersegment calls while the other application's resource fork is open. The discussion of this issue in relation to `FSpOpenResFile` (page 1-60) also applies to `HOpenResFile`.

#### ASSEMBLY-LANGUAGE INFORMATION

A handle to the resource map for the most recently opened resource fork is stored in the global variable `TopMapHndl`.

## CHAPTER 1

### Resource Manager

#### RESULT CODES

noErr	0	No error
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename or volume name (perhaps zero length)
eofErr	-39	End of file
tmfoErr	-42	Too many files open
fnfErr	-43	File not found
opWrErr	-49	File already open with write permission
permErr	-54	Attempt to open locked file for writing
extFSErr	-58	Volume belongs to an external file system
memFullErr	-108	Not enough room in heap zone
dirNFErr	-120	Directory not found
mapReadErr	-199	Map inconsistent with operation

#### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about `permission` parameter constants and the `OpenRF` function, see *Inside Macintosh: Files*.

## OpenRFPPerm

---

If the `FSpOpenResFile` and `HOpenResFile` functions are not available, you can use the `OpenRFPPerm` function to open a file's resource fork.

```
FUNCTION OpenRFPPerm (fileName: Str255; vRefNum: Integer;  
                    permission: SignedByte): Integer;
```

`fileName`     The name of the file whose resource fork is to be opened.

`vRefNum`     The volume reference number or directory ID for the volume or directory in which the file is located.

`permission`     A constant for one of the read/write permission combinations.

#### DESCRIPTION

The `OpenRFPPerm` function opens the resource fork of the file with the name specified by the `fileName` parameter in the directory or volume specified by the `vRefNum` parameter. It also makes this file the current resource file.

In addition to opening the resource fork for the file with the specified name, `OpenRFPPerm` lets you specify in the `permission` parameter the read/write permission of the resource fork the first time it is opened.



You can use the `OpenRFPPerm` function if the `FSpOpenResFile` function is not available. You can determine whether `FSpOpenResFile` is available by calling the `Gestalt` function with the `gestaltFSAttr` selector code. The `OpenRFPPerm` is an earlier version of the `HOpenResFile` function.

You can specify the access path permission for the resource fork by setting the permission parameter to one of these constants:

```
CONST
    fsCurPerm      = 0; {whatever is currently allowed}
    fsRdPerm       = 1; {read-only permission}
    fsWrPerm       = 2; {write permission}
    fsRdWrPerm     = 3; {exclusive read/write permission}
    fsRdWrShPerm  = 4; {shared read/write permission}
```

See page 1-59 for information about specifying access path permission with `FSpOpenResFile`. The same information applies to `OpenRFPPerm`.

The Resource Manager reads the resource map from the resource fork for the specified file into memory. It also reads into memory every resource in the resource fork whose `resPreload` attribute is set.

The `OpenRFPPerm` function returns a file reference number for the file whose resource fork it has opened. You can use this file reference number to refer to the file in other Resource Manager routines. If the file's resource fork is already open, `OpenRFPPerm` returns the file reference number but does not make that file the current resource file.

If the `OpenRFPPerm` function fails to open the specified file's resource fork (because there's no file with the given name or because there are permission problems), it returns -1 as the file reference number. Use the `ResError` function to determine what kind of error occurred.

You don't have to call `OpenRFPPerm` to open the System file's resource fork or an application file's resource fork. These files are opened automatically when the system and the application start up, respectively. To get the file reference number for your application, call the `CurResFile` function after the application starts up and before you open the resource forks for any other files.

The `OpenRFPPerm` function checks that the information in the resource map is internally consistent. If it isn't, `ResError` returns the result code `mapReadErr`.

To open a resource fork just for block-level operations, such as copying files without reading the resource map into memory, use the File Manager function `OpenRF`.

#### SPECIAL CONSIDERATIONS

The `OpenRFPPerm` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

It's possible to create multiple, unique, read-only access paths to a resource fork using `OpenRFPPerm`; however, you should avoid doing so. See page 1-60 for discussion of this issue in relation to `FSpOpenResFile`; `OpenRFPPerm` works the same way.

## CHAPTER 1

### Resource Manager

Versions of system software before System 7 do not allow you to use `OpenRFPPerm` to open a second access path, with write access, to a resource fork. In this case, `OpenRFPPerm` returns the reference number already assigned to the file.

If you want to open the resource fork for another application (or any resource fork other than your application's that includes 'CODE' resources), you must bracket your calls to `OpenRFPPerm` with calls to `SetResLoad` with the `load` parameter set to `FALSE` and then to `TRUE`. You must also avoid making intersegment calls while the other application's resource fork is open. The discussion of this issue in relation to `FSpOpenResFile` (page 1-60) also applies to `OpenRFPPerm`.

#### ASSEMBLY-LANGUAGE INFORMATION

A handle to the resource map for the most recently opened resource fork is stored in the global variable `TopMapHndl`.

#### RESULT CODES

<code>noErr</code>	0	No error
<code>nsvErr</code>	-35	No such volume
<code>ioErr</code>	-36	I/O error
<code>bdNamErr</code>	-37	Bad filename or volume name (perhaps zero length)
<code>eofErr</code>	-39	End of file
<code>tmfoErr</code>	-42	Too many files open
<code>fnfErr</code>	-43	File not found
<code>opWrErr</code>	-49	File already open with write permission
<code>permErr</code>	-54	Attempt to open locked file for writing
<code>extFSerr</code>	-58	Volume belongs to an external file system
<code>memFullErr</code>	-108	Not enough room in heap zone
<code>dirNFErr</code>	-120	Directory not found
<code>mapReadErr</code>	-199	Map inconsistent with operation

#### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about `permission` parameter constants and the `OpenRF` function, see *Inside Macintosh: Files*.

## OpenResFile

---

If the `FSpOpenResFile` function is not available, you can use the `OpenResFile` function to open a resource fork.

```
FUNCTION OpenResFile (fileName: Str255): Integer;
```

`fileName`    The name of the file whose resource fork is to be opened.

**DESCRIPTION**

The `OpenResFile` function opens the resource fork of the file with the name specified by the `fileName` parameter in the application's default directory—that is, the directory in which the application is located. It also makes this file the current resource file.

Like the `OpenRFPPerm` function, the `OpenResFile` function takes a filename and opens the resource fork for the file with that name. Unlike `OpenRFPPerm`, `OpenResFile` does not let you specify the read/write permission of the resource fork the first time it is opened. The `OpenResFile` function is an earlier version of the `OpenRFPPerm` function.

If it finds the specified file in your application's default directory, `OpenResFile` reads the file's resource map into memory and returns a file reference number for the file. It also reads into memory every resource in the resource fork whose `resPreload` attribute is set.

You can use the file reference number returned by `OpenResFile` to refer to the file in other Resource Manager routines. If the file's resource fork is already open, `OpenResFile` returns the file reference number but does not make that file the current resource file.

If the `OpenResFile` function fails to open the specified file's resource fork (for instance, because there's no file with the given name), it returns `-1` as the file reference number. Use the `ResError` function to determine what kind of error occurred.

You don't have to call `OpenResFile` to open the System file's resource fork or an application file's resource fork. These resource forks are opened automatically when the system and the application start up, respectively. To get the file reference number for your application, call the `CurResFile` function after the application starts up and before you open the resource forks for any other files.

The `OpenResFile` function checks that the information in the resource map is internally consistent. If it isn't, `ResError` returns the result code `mapReadErr`.

To open a resource fork just for block-level operations, such as copying files without reading the resource map into memory, use the File Manager function `OpenRF`.

**SPECIAL CONSIDERATIONS**

The `OpenResFile` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

If you want to open the resource fork for another application (or any resource fork other than your application's that includes 'CODE' resources), you must bracket your calls to `OpenResFile` with calls to `SetResLoad` with the `load` parameter set to `FALSE` and then to `TRUE`. You must also avoid making intersegment calls while the other application's resource fork is open. The discussion of this issue in relation to `FSpOpenResFile` (page 1-60) also applies to `OpenResFile`.

**ASSEMBLY-LANGUAGE INFORMATION**

A handle to the resource map for the most recently opened resource fork is stored in the global variable `TopMapHndl`.

**RESULT CODES**

noErr	0	No error
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename or volume name (perhaps zero length)
eofErr	-39	End of file
tmfoErr	-42	Too many files open
fnfErr	-43	File not found
opWrErr	-49	File already open with write permission
permErr	-54	Attempt to open locked file for writing
extFSErr	-58	Volume belongs to an external file system
memFullErr	-108	Not enough room in heap zone
dirNFErr	-120	Directory not found
mapReadErr	-199	Map inconsistent with operation

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

## Getting and Setting the Current Resource File

---

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 file where they should begin. In general, the current resource file is the last one whose resource fork your application opened unless you specify otherwise.

Two routines work specifically with the current resource file: `CurResFile` and `UseResFile`. The `CurResFile` function tells you which of the files whose resource forks are currently open is the current resource file. The `UseResFile` procedure sets the current resource file.

The `HomeResFile` function gets the file reference number associated with a particular resource.

## CurResFile

---

You can use the `CurResFile` function to get the file reference number of the current resource file.

```
FUNCTION CurResFile: Integer;
```

**DESCRIPTION**

The `CurResFile` function returns the file reference number associated with the current resource file. You can call this function when your application starts up (before opening the resource fork of any other file) to get the file reference number of your application's resource fork.

If the current resource file is the System file, `CurResFile` returns the actual file reference number. You can use this number or 0 with routines that take a file reference number for the System file. All Resource Manager routines recognize both 0 and the actual file reference number as referring to the System file.

#### ASSEMBLY-LANGUAGE INFORMATION

The current resource file's reference number is stored in the global variable `CurMap`.

#### RESULT CODE

`noErr`    0    No error

#### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `CurResFile` function, see Listing 1-8 on page 1-29.

## UseResFile

---

You can use the `UseResFile` procedure to set the current resource file.

```
PROCEDURE UseResFile (refNum: Integer);
```

`refNum`        The file reference number for a resource fork.

#### DESCRIPTION

The `UseResFile` procedure searches the list of files whose resource forks have been opened for the file specified by the `refNum` parameter. If the specified file is found, the Resource Manager sets the current resource file to the specified file. If there's no resource fork open for a file with that reference number, `UseResFile` does nothing. To set the current resource file to the System file, use 0 for the `refNum` parameter.

Open resource forks are arranged as a linked list with the most recently opened resource fork at the beginning. When searching open resource forks, the Resource Manager starts with the most recently opened file. You can call the `UseResFile` procedure to set the current resource file to a file opened earlier, and thereby start subsequent searches with the specified file. In this way, you can cause any files higher in the resource chain to be left out of subsequent searches.

When a new resource fork is opened, this action overrides previous calls to `UseResFile` and the entire list is searched. For example, if five resource forks are opened in the order R0, R1, R2, R3, and R4, the search order is R4-R3-R2-R1-R0. Calling `UseResFile(R2)` changes the search order to R2-R1-R0; R4 and R3 are not searched. When the resource fork of a new file (R5) is opened, the search order becomes R5-R4-R3-R2-R1-R0.

## CHAPTER 1

### Resource Manager

You typically call `CurResFile` to get and save the current resource file, `UseResFile` to set the current resource file to the desired file, then (after you are finished using the resource) `UseResFile` to restore the current resource file to its previous value. Calling `UseResFile(0)` causes the Resource Manager to search only the System file's resource map. This is useful if you no longer wish to override a system resource with one by the same name in your application's resource fork.

#### SPECIAL CONSIDERATIONS

The `FSpOpenResFile`, `HOpenResFile`, and `OpenResFile` functions, which also set the current resource file, override previous calls to `UseResFile`.

#### ASSEMBLY-LANGUAGE INFORMATION

The settings of the system global variables `RomMapInsert` and `TmpResLoad` affect resource search order. These global variables determine whether the Resource Manager searches ROM-resident resources before the System file's resources.

The Resource Manager normally searches ROM resources only when you use the `RGetResource` function to get a handle to the resource, and even then only after it searches the System file's resource fork. To search for a resource in ROM before searching the System file's resource fork, your application must first alter the resource search order by inserting the ROM resource map in front of the System file's resource map.

When the value of the system global variable `RomMapInsert` is `TRUE`, the Resource Manager inserts the ROM resource map before the System file's resource map for the next call only (including any Resource Manager routine that gets a resource, not just `RGetResource`). When the value of `RomMapInsert` is `TRUE`, the adjacent variable `TmpResLoad` determines whether the value of the global variable `ResLoad` is considered `TRUE` or `FALSE`, overriding the actual value of `ResLoad` for the next call only. The values of the `RomMapInsert` and `TmpResLoad` variables are cleared after each call to a Resource Manager routine.

You can use two global constants to set these variables in tandem. Set the system global variable `RomMapInsert` to the global constant `mapTrue` to insert the ROM resource map with `SetResLoad(TRUE)`. Set the system global variable `RomMapInsert` to the global constant `mapFalse` to insert the ROM resource map with `SetResLoad(FALSE)`.

#### RESULT CODES

<code>noErr</code>	0	No error
<code>resFNotFound</code>	-193	Resource file not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `UseResFile` procedure, see Listing 1-8 on page 1-29.

For descriptions of the `FSpOpenResFile`, `HOpenResFile`, and `OpenResFile` functions, see page 1-58 through page 1-66. For a description of the `SetResLoad` procedure, see page 1-79.

**HomeResFile**

---

To get the file reference number associated with a particular resource, use the `HomeResFile` function.

```
FUNCTION HomeResFile (theResource: Handle): Integer;
```

`theResource`

A handle to a resource.

**DESCRIPTION**

Given a handle to a resource, the `HomeResFile` function returns the file reference number for the resource fork containing the specified resource. If the given handle isn't a handle to a resource, `HomeResFile` returns `-1`, and the `ResError` function returns the result code `resNotFound`. If `HomeResFile` returns `0`, the resource is in the System file's resource fork. If `HomeResFile` returns `1`, the resource is ROM-resident.

**RESULT CODES**

<code>noErr</code>	<code>0</code>	No error
<code>resNotFound</code>	<code>-192</code>	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

**Reading Resources Into Memory**

---

The routines described in this section allow your application to read resource data into memory. The `GetResource` and `Get1Resource` functions get a resource specified by a resource type and a resource ID. The `GetNamedResource` and `Get1NamedResource` functions get a resource specified by name. The `RGetResource` function searches the ROM-resident resources as well as the open resource forks.

The `SetResLoad` procedure enables and disables automatic loading of resource data into memory for routines that return handles to resources, and the `LoadResource` procedure reads resource data into memory for a purged resource or after you've called `SetResLoad` with the `load` parameter set to `FALSE`.

When your application requests a resource, the Resource Manager normally looks in the current resource file's resource map in memory. If it can't find an entry for the specified resource, the Resource Manager searches the resource maps for each open resource fork in the reverse order that the resource forks were opened. If it can't find an entry for the specified resource in any of these resource maps, the Resource Manager searches your application's resource map. If it can't find an entry for the specified resource in your application's resource map, the Resource Manager searches the resource map for the System file.

The Resource Manager determines whether or not to load the specified resource into memory according to the entry for that resource in the resource map. If the resource's resource map entry contains a valid handle, the Resource Manager returns that handle. If the value of the handle is `NIL`, the Resource Manager reads the resource data into memory.

Before reading the resource data into memory, the Resource Manager calls the Memory Manager to allocate a relocatable block for the resource data. The Memory Manager allocates the block, assigns a master pointer to the block, and returns to the Resource Manager a pointer to the master pointer. The Resource Manager then installs this handle in the resource map and also returns a handle to the resource.

If the resource's resource map entry contains an empty handle (a handle whose master pointer is set to `NIL`) and the value of the system global variable `ResLoad` is `TRUE`, the Resource Manager routines that get resources reallocate the resource's handle and read the resource data from disk back into memory.

**IMPORTANT**

In certain situations, a Resource Manager routine can return an empty handle (a handle whose master pointer is set to `NIL`). For instance, if you've called `SetResLoad` with the `load` parameter set to `FALSE` and the resource data isn't already in memory, and then you call the `GetResource` function (or any of the other Resource Manager routines that get a resource), the Resource Manager routine returns an empty handle (a handle whose master pointer is set to `NIL`). This can also happen if you read resource data for a purgeable resource into memory and then call `SetResLoad` with the `load` parameter set to `FALSE`. If the resource data is later purged, when you call `GetResource` (or other routines that get a resource), the Resource Manager returns an empty handle. You should test for an empty handle in these situations. To make the handle a valid handle to resource data in memory, you can call the `LoadResource` procedure. ♦



## GetResource

---

You can use the `GetResource` function to get resource data for a resource specified by resource type and resource ID.

```
FUNCTION GetResource (theType: ResType; theID: Integer): Handle;
```

`theType`      A resource type.

`theID`        An integer that uniquely identifies a resource of the specified type.

### DESCRIPTION

The `GetResource` function searches the resource maps in memory for the resource specified by the parameters `theType` and `theID`. The resource maps in memory, which represent all the open resource forks, are arranged as a linked list. When searching this list, `GetResource` starts with the current resource file and progresses through the list (that is, searching the resource maps in reverse order of opening) until it finds the resource's entry in one of the resource maps.

If the `GetResource` function finds the specified resource entry in one of the resource maps and the entry contains a valid handle, it returns that handle. If the entry contains a handle whose value is `NIL`, and if you haven't called `SetResLoad` with the `load` parameter set to `FALSE`, `GetResource` attempts to read the resource into memory.

If `GetResource` can't find the resource data, it returns `NIL`, and `ResError` returns the result code `resNotFound`. The `GetResource` function also returns `NIL` if the resource data to be read into memory won't fit, in which case `ResError` returns an appropriate Memory Manager result code. If you call `GetResource` with a resource type that can't be found in any of the resource maps of the open resource forks, the function returns `NIL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NIL`.

You can change the resource map search order by calling the `UseResFile` procedure before `GetResource`.

### SPECIAL CONSIDERATIONS

Calling `GetResource` may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

### RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For examples of the use of `GetResource`, see page 1-18 through page 1-24.

To include ROM-resident system resources in the Resource Manager's search of the resource maps of open resource forks, use the `RGetResource` function as described on page 1-78.

For information about the `UseResFile` and `SetResLoad` procedures, see page 1-69 and page 1-79, respectively.

**Get1Resource**

---

You can use the `Get1Resource` function to get resource data for a resource in the current resource file.

```
FUNCTION Get1Resource (theType: ResType; theID: Integer): Handle;
```

`theType`     A resource type.

`theID`       An integer that uniquely identifies a resource of the specified type.

**DESCRIPTION**

The `Get1Resource` function searches the current resource file's resource map in memory for the resource specified by the `theType` and `theID` parameters. If `Get1Resource` finds an entry for the resource in the current resource file's resource map and the entry contains a valid handle, it returns that handle. If the entry contains a handle whose value is `NIL`, and if you haven't called `SetResLoad` with the `load` parameter set to `FALSE`, `Get1Resource` attempts to read the resource into memory.

If `Get1Resource` can't find the resource data, it returns `NIL`, and `ResError` returns the result code `resNotFound`. The `Get1Resource` function also returns `NIL` if the resource data to be read into memory won't fit, in which case `ResError` returns an appropriate Memory Manager result code.

If you call `Get1Resource` with a resource type that can't be found in the resource map of the current resource file, the function returns `NIL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NIL`.

You can change the resource map search order by calling the `UseResFile` procedure before `Get1Resource`.

**SPECIAL CONSIDERATIONS**

Calling `Get1Resource` may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For examples of the use of the `Get1Resource` function, see Listing 1-8 on page 1-29 and Listing 1-9 on page 1-32.

To include ROM-resident system resources in the Resource Manager's search of the resource maps for open resource forks, use the `RGetResource` function as described on page 1-78.

For information about the `UseResFile` and `SetResLoad` procedures, see page 1-69 and page 1-79, respectively.

## **GetNamedResource**

---

You can use the `GetNamedResource` function to get a named resource.

```
FUNCTION GetNamedResource (theType: ResType; name: Str255)
                        : Handle;
```

`theType`     A resource type.

`name`        A name that uniquely identifies a resource of the specified type. Strings passed in this parameter are case-sensitive.

**DESCRIPTION**

The `GetNamedResource` function searches the resource maps in memory for the resource specified by the parameters `theType` and `name`. The resource maps in memory, which represent all the open resource forks, are arranged as a linked list. When `GetNamedResource` searches this list, it starts with the current resource file and progresses through the list in order (that is, in reverse chronological order in which the resource forks were opened) until it finds the resource's entry in one of the resource maps.

## CHAPTER 1

### Resource Manager

If `GetNamedResource` finds the specified resource entry in one of the resource maps and the entry contains a valid handle, the function returns that handle. If the entry contains a handle whose value is `NIL`, and if you haven't called `SetResLoad` with the load parameter set to `FALSE`, `GetNamedResource` attempts to read the resource into memory.

If the `GetNamedResource` function can't find the resource data, it returns `NIL`, and `ResError` returns the result code `resNotFound`. The `GetNamedResource` function also returns `NIL` if the resource data to be read into memory won't fit, in which case `ResError` returns an appropriate Memory Manager result code. If you call `GetNamedResource` with a resource type that can't be found in any of the resource maps of the open resource forks, the function returns `NIL` as well, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NIL`.

You can change the resource map search order by calling the `UseResFile` procedure before `GetNamedResource`.

#### SPECIAL CONSIDERATIONS

The `GetNamedResource` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

#### RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

#### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

To include ROM-resident system resources in the Resource Manager's search of the resource maps for open resource forks, use the `RGetResource` function as described on page 1-78.

For information about the `UseResFile` and `SetResLoad` procedures, see page 1-69 and page 1-79, respectively.

## Get1NamedResource

---

You can use the `Get1NamedResource` function to get a named resource in the current resource file.

```
FUNCTION Get1NamedResource (theType: ResType; name: Str255)
                          : Handle;
```

`theType`      A resource type.

`name`          A name that uniquely identifies a resource of the specified type.

### DESCRIPTION

The `Get1NamedResource` function searches the current resource file's resource map in memory for the resource specified by the parameters `theType` and `name`. If `Get1NamedResource` finds an entry for the resource in the current resource file's resource map and the entry contains a valid handle, the function returns that handle. If the entry contains a handle whose value is `NIL`, and if you haven't called `SetResLoad` with the `load` parameter set to `FALSE`, `Get1NamedResource` attempts to read the resource into memory.

If it can't find the resource data, `Get1NamedResource` returns `NIL`, and `ResError` returns the result code `resNotFound`. The `Get1NamedResource` function also returns `NIL` if the resource data to be read into memory won't fit, in which case `ResError` returns an appropriate Memory Manager result code.

If you call `Get1NamedResource` with a resource type that can't be found in the resource map of the current resource file, the function returns `NIL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NIL`.

You can change the search order by calling the `UseResFile` procedure before `Get1NamedResource`.

### SPECIAL CONSIDERATIONS

The `Get1NamedResource` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

### RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

To include ROM-resident system resources in the Resource Manager's search of the resource maps for open resource forks, use the `RGetResource` function, described next.

For information about the `UseResFile` and `SetResLoad` procedures, see page 1-69 and page 1-79, respectively.

**RGetResource**

---

You can use the `RGetResource` function to get resource data for a resource and include ROM-resident system resources in the Resource Manager's search of resource maps.

```
FUNCTION RGetResource (theType: ResType; theID: Integer): Handle;
```

`theType`      A resource type.

`theID`        An integer that uniquely identifies a resource of the specified type.

**DESCRIPTION**

The `RGetResource` function searches the resource maps in memory for the resource specified by the parameters `theType` and `theID`. The resource maps in memory, which represent all open resource forks, are arranged as a linked list. The `RGetResource` function first uses `GetResource` to search this list. The `GetResource` function starts with the current resource file and progresses through the list in order (that is, in reverse chronological order in which the resource forks were opened) until it finds the resource's entry in one of the resource maps. If `GetResource` doesn't find the specified resource in its search of the resource maps of open resource forks (which includes the System file's resource fork), `RGetResource` sets the global variable `RomMapInsert` to `TRUE`, then calls `GetResource` again. In response, `GetResource` performs the same search, but this time it looks in the resource map of the ROM-resident resources before searching the resource map of the System file.

If `RGetResource` finds the specified resource entry in one of the resource maps and the entry contains a valid handle, the function returns that handle. If the entry contains a handle whose value is `NIL`, and if you haven't called `SetResLoad` with the `load` parameter set to `FALSE`, `RGetResource` attempts to read the resource into memory.

If it can't find the resource data, `RGetResource` returns `NIL`, and `ResError` returns the result code `resNotFound`. The `RGetResource` function also returns `NIL` if the resource data to be read into memory won't fit, in which case `ResError` returns an appropriate Memory Manager result code. If you call `RGetResource` with a resource type that can't be found in any of the resource maps of the open resource forks, the function returns `NIL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NIL`.

**SPECIAL CONSIDERATIONS**

The `RGetResource` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For more information, see “Inserting the ROM Resource Map” beginning on page 1-134.

For a description of the `UseResFile` procedure, see page 1-69. The `SetResLoad` procedure is described next.

**SetResLoad**

---

You can use the `SetResLoad` procedure to enable and disable automatic loading of resource data into memory for routines that return handles to resources.

```
PROCEDURE SetResLoad (load: Boolean);
```

`load`      A Boolean value that determines whether Resource Manager routines should read resource data into memory. If you set this parameter to `TRUE`, Resource Manager routines that return handles will, during subsequent calls, automatically read resource data into memory if it is not already in memory; if you set this parameter to `FALSE`, Resource Manager routines will not automatically read resource data into memory.

**DESCRIPTION**

Routines that return handles to resources normally read the resource data into memory if it's not already there. The default setting (`load = TRUE`) maintains this state. If the `load` parameter is set to `FALSE`, routines that return handles to resources will not, during subsequent calls, 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. In addition, when first opening a resource fork the Resource Manager won't load into memory resources whose `resPreload` attribute is set.

You can use the `SetResLoad` procedure 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`, call the `LoadResource` procedure, which is described next.

▲ **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 expect this value to be `TRUE`, and some routines won't work if resources are not loaded automatically. ▲

**ASSEMBLY-LANGUAGE INFORMATION**

The current value of `SetResLoad` is stored in the global variable `ResLoad`.

**RESULT CODE**

`noErr`    0    No error

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For more information about the global variable `ResLoad`, see “Inserting the ROM Resource Map” beginning on page 1-134.

## LoadResource

---

You can use the `LoadResource` procedure to get resource data after you've called `SetResLoad` with the `load` parameter set to `FALSE` or when the resource is purgeable.

```
PROCEDURE LoadResource (theResource: Handle);
```

`theResource`

A handle to a resource.

**DESCRIPTION**

Given a handle to a resource, `LoadResource` reads the resource data into memory. If the resource is already in memory, or if the `theResource` parameter doesn't contain a handle to a resource, then `LoadResource` does nothing. To determine whether either of these situations occurred, call `ResError`. If the resource is already in memory `ResError` returns `noErr`; if the handle is not a handle to a resource, `ResError` returns `resNotFound`.



**SPECIAL CONSIDERATIONS**

If you've changed the resource data for a purgeable resource and the resource is purged before being written to the file, the changes will be lost. In this case, `LoadResource` rereads the original resource from the file's resource fork. You should use `ChangedResource` or `SetResPurge` before calling `LoadResource` to ensure that changes made to purgeable resources are written to the resource fork.

**ASSEMBLY-LANGUAGE INFORMATION**

The `LoadResource` procedure preserves all registers.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For information about the `SetResLoad`, `ChangedResource`, and `SetResPurge` procedures, see page 1-79, page 1-88, and page 1-94, respectively.

## Getting and Setting Resource Information

---

The Resource Manager provides four routines that allow you to get and set information about resources. The `GetResInfo` procedure returns the resource ID, resource type, and resource name for a specified resource. The `SetResInfo` procedure sets the resource name and resource ID for a specified resource. The `GetResAttrs` function returns a resource's attributes, and the `SetResAttrs` function sets a resource's attributes.

### GetResInfo

---

You can use the `GetResInfo` procedure to get a resource's resource ID, resource type, and resource name.

```
PROCEDURE GetResInfo (theResource: Handle; VAR theID: Integer;
                    VAR theType: ResType; VAR name: Str255);
```

`theResource`

A handle to a resource.

`theID`

`GetResInfo` returns the resource ID of the specified resource in this parameter.

## CHAPTER 1

### Resource Manager

`theType`      `GetResInfo` returns the resource type of the specified resource in this parameter.

`name`            `GetResInfo` returns the name of the specified resource in this parameter.

#### DESCRIPTION

Given a handle to a resource, the `GetResInfo` procedure returns the resource's resource ID, resource type, and resource name. If the handle isn't a valid handle to a resource, `GetResInfo` does nothing; to determine whether this has occurred, call `ResError`.

#### RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

#### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

To set a resource's ID, resource type, or resource name, use the `SetResInfo` procedure. It is described next.

## SetResInfo

---

You can use the `SetResInfo` procedure to change the name and resource ID of a resource.

```
PROCEDURE SetResInfo (theResource: Handle; theID: Integer;
                      name: Str255);
```

`theResource`      A handle to a resource.

`theID`            The new resource ID.

`name`            The new name or an empty string to preserve the resource name.

#### DESCRIPTION

Given a handle to a resource, `SetResInfo` changes the resource ID and the resource name of the specified resource to the values given in `theID` and `name`. If you pass an empty string for the `name` parameter, the resource name is not changed. The `SetResInfo` procedure changes the information in the resource map in memory, not in the resource file itself.

**▲ WARNING**

Do not change a system resource's resource ID or name. Other applications may already access the resource and may not work properly if you change the resource ID, resource name, or both. ▲

If the parameter `theResource` doesn't contain a handle to an existing resource, `SetResInfo` does nothing, and `ResError` returns the result code `resNotFound`. If the resource map becomes too large to fit in memory (for example, after an unnamed resource is given a name), `SetResInfo` does nothing, and `ResError` returns an appropriate Memory Manager result code. The same is true if the resource data in memory can't be written to the resource fork (for example, because the disk is full). If the `resProtected` attribute is set for the resource, `SetResInfo` does nothing, and `ResError` returns the result code `resAttrErr`.

If you want to write changes to the resource map on disk after updating the resource map in memory, call the `ChangedResource` procedure for the same resource after you call `SetResInfo`.

**IMPORTANT**

Even if you don't call `ChangedResource` after using `SetResInfo` to change the name and resource ID of a resource, the change may be written to disk when the Resource Manager updates the resource fork. If you call `ChangedResource` for *any* resource in the same resource fork, or if you add or remove a resource, the Resource Manager writes the entire resource map to disk after a call to `UpdateResFile` or when your application terminates. In these cases, all changes to resource information in the resource map become permanent. If you want any of the changes to be temporary, you should restore the original information before the resource is updated. ▲

**SPECIAL CONSIDERATIONS**

The `SetResInfo` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>resAttrErr</code>	-198	Attribute does not permit operation

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `ChangedResource` and `UpdateResFile` procedures, see page 1-88 and page 1-92, respectively.

## GetResAttrs

---

You can use the `GetResAttrs` function to get a resource's attributes.

```
FUNCTION GetResAttrs (theResource: Handle): Integer;
```

`theResource`

A handle to a resource.

### DESCRIPTION

Given a handle to a resource, the `GetResAttrs` function returns the resource's attributes as recorded in its entry in the resource map in memory. If the value of the `theResource` parameter isn't a handle to a valid resource, `GetResInfo` does nothing, and the `ResError` function returns the result code `resNotFound`.

The `GetResAttrs` function returns the resource's attributes in the low-order byte of the function result. Each attribute is identified by a specific bit in the low-order byte. If the bit corresponding to an attribute contains 1, then that attribute is set; if the bit contains 0, then that attribute is not set. You can use these constants to refer to each attribute:

CONST

```
resSysHeap      = 64;    {set if read into system heap}
resPurgeable    = 32;    {set if purgeable}
resLocked       = 16;    {set if locked}
resProtected    = 8;     {set if protected}
resPreload      = 4;     {set if to be preloaded}
resChanged      = 2;     {set if to be written to resource fork}
```

The `resSysHeap` attribute indicates whether the resource is read into the system heap (`resSysHeap` attribute is set to 1) or your application's heap (`resSysHeap` attribute is set to 0).

If the `resPurgeable` attribute is set to 1, the resource is purgeable; if it's 0, the resource is nonpurgeable.

Because a locked resource is nonrelocatable and nonpurgeable, the `resLocked` attribute overrides the `resPurgeable` attribute. If the `resLocked` attribute is 1, the resource is nonpurgeable regardless of whether `resPurgeable` is set. If it's 0, the resource is purgeable or nonpurgeable depending on the value of the `resPurgeable` attribute.

If the `resProtected` attribute is set to 1, your application can't use Resource Manager routines to change the resource ID or resource name, modify the resource contents, or remove the resource from its resource fork. However, you can use the `SetResAttrs` procedure to remove this protection.

If the `resPreload` attribute is set to 1, the Resource Manager reads the resource's resource data into memory immediately after opening its resource fork. You can use this setting to make multiple resources available for your application as soon as possible,

rather than reading each one into memory individually. If both the `resPreload` attribute and the `resLocked` attribute are set, the Resource Manager loads the resource as low in the heap as possible.

If the `resChanged` attribute is set to 1, the resource has been changed; if it's 0, the resource hasn't been changed. This attribute is used only while the resource map is in memory. The `resChanged` attribute must be 0 in the resource fork on disk.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For more information about resource attributes, see “The Resource Map” beginning on page 1-8.

To change a resource's attributes in the resource map in memory, use the `SetResAttrs` procedure. It is described next.

**SetResAttrs**

---

You can use the `SetResAttrs` procedure to change a resource's attributes in the resource map in memory.

```
PROCEDURE SetResAttrs (theResource: Handle; attrs: Integer);
```

`theResource`

A handle to a resource.

`attrs`

The resource attributes to set.

**DESCRIPTION**

Given a handle to a resource, `SetResAttrs` changes the resource attributes of the resource to those specified in the `attrs` parameter. The `SetResAttrs` procedure changes the information in the resource map in memory, not in the file on disk. The `resProtected` attribute changes immediately. Other attribute changes take effect the next time the specified resource is read into memory but are not made permanent until the Resource Manager updates the resource fork.

If the value of the parameter `theResource` isn't a valid handle to a resource, `SetResAttrs` does nothing, and the `ResError` function returns the result code `resNotFound`.

## Resource Manager

Each attribute is identified by a specific bit in the low-order byte of a word. If the bit corresponding to an attribute contains 1, then that attribute is set; if the bit contains 0, then that attribute is not set. You can use these constants to specify each attribute:

```
CONST
    resSysHeap      = 64;    {set if read into system heap}
    resPurgeable   = 32;    {set if purgeable}
    resLocked      = 16;    {set if locked}
    resProtected   = 8;     {set if protected}
    resPreload     = 4;     {set if to be preloaded}
    resChanged     = 2;     {set if to be written to resource fork}
```

The `resSysHeap` attribute determines whether the resource is read into your application's heap (`resSysHeap` attribute set to 0) or the system heap (`resSysHeap` attribute set to 1). You should set this bit to 0 for your application's resources. Note that if you do set the `resSysHeap` attribute to 1 and the resource is too large for the system heap, the bit is cleared and the resource is read into the application heap.

Set the `resPurgeable` attribute to 1 to make the resource purgeable; you can set it to 0 to make the resource nonpurgeable. However, do not use `SetResAttrs` to make a purgeable resource nonpurgeable.

Because a locked resource is nonrelocatable and nonpurgeable, the `resLocked` attribute overrides the `resPurgeable` attribute. If you set the `resLocked` attribute to 1, the resource is nonpurgeable regardless of whether or not you set `resPurgeable`. If you set the `resLocked` attribute to 0, the resource is purgeable or nonpurgeable depending on the value of the `resPurgeable` attribute.

If you set the `resProtected` attribute to 1, your application can't use Resource Manager routines to change the resource ID or resource name, modify the resource contents, or remove the resource from its resource fork. If you set the `resProtected` attribute to 0, you remove this protection. Note that this attribute change takes effect immediately.

If you set the `resPreload` attribute to 1, the Resource Manager reads the resource's resource data into memory immediately after opening its resource fork. You can use this setting to make multiple resources available for your application as soon as possible, rather than reading each one into memory separately.

The `resChanged` attribute indicates whether or not the resource has been changed; do not use `SetResAttrs` to set the `resChanged` attribute. Be sure the `attrs` parameter passed to `SetResAttrs` doesn't change the current setting of this attribute. To determine the attribute's current setting, call the `GetResAttrs` function. To set the `resChanged` attribute, call the `ChangedResource` procedure. Note that the `resChanged` attribute is used only while the resource map is in memory. The `resChanged` attribute must be 0 in the resource fork on disk.

If you want the Resource Manager to write the modified resource map to disk after a subsequent call to `UpdateResFile` or when your application terminates, call the `ChangedResource` procedure after you call `SetResAttrs`.

▲ **WARNING**

Do not use `SetResAttrs` to change a purgeable resource. If you make a purgeable resource nonpurgeable by setting the `resPurgeable` attribute with `SetResAttrs`, the resource doesn't become nonpurgeable until the next time the specified resource is read into memory. Thus, the resource might be purged while you're changing it. ▲

**SPECIAL CONSIDERATIONS**

The `SetResAttrs` procedure does not return an error if you are setting the attributes of a resource in a resource file that has a read-only resource map. To find out whether this is the case, use `GetResFileAttrs`.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For more information about resource attributes, see “The Resource Map” beginning on page 1-8.

For a description of the `GetResFileAttrs` function, see page 1-116. To mark a resource as changed, use the `ChangedResource` procedure, described next.

## Modifying Resources

---

The Resource Manager provides two routines that change the `resChanged` attribute of a specified resource. The `ChangedResource` procedure allows you to indicate that a resource in memory has been changed, and the `AddResource` procedure allows you to add a new resource to a resource map.

If the `resChanged` attribute for a resource has been set and your application calls `UpdateResFile` or quits, the Resource Manager writes both the entire resource map and the resource data for that resource to the resource fork of the corresponding file on disk. If the `resChanged` attribute has been set and your application calls `WriteResource`, the Resource Manager writes only the resource's data to disk.

## ChangedResource

---

If you've changed a resource's data or changed an entry in a resource map, you can use the `ChangedResource` procedure to set a flag in the resource's resource map entry in memory to show that you've made changes.

```
PROCEDURE ChangedResource (theResource: Handle);
```

```
theResource
```

A handle to a resource.

### DESCRIPTION

Given a handle to a resource, the `ChangedResource` procedure sets the `resChanged` attribute for that resource in the resource map in memory. If the `resChanged` attribute for a resource has been set and your application calls `UpdateResFile` or quits, the Resource Manager writes the resource data for that resource (and for all other resources whose `resChanged` attribute is set) and the entire resource map to the resource fork of the corresponding file on disk. If the `resChanged` attribute for a resource has been set and your application calls `WriteResource`, the Resource Manager writes only the resource data for that resource to disk.

If you change information in the resource map with a call to `SetResInfo` or `SetResAttrs` and then call `ChangedResource` and `UpdateResFile`, the Resource Manager still writes both the resource map and the resource data to disk. If you want any of your changes to the resource map or resource data to be temporary, you must restore the original information before the Resource Manager updates the resource fork on disk.

After writing a resource to disk, the Resource Manager clears the resource's `resChanged` attribute in the appropriate entry of the resource map in memory.

If the given handle isn't a handle to a resource, if the modified resource data can't be written to the resource fork, or if the `resProtected` attribute is set for the modified resource, `ChangedResource` does nothing. To find out whether any of these errors occurred, call `ResError`.

When your application calls `ChangedResource`, the Resource Manager attempts to reserve enough disk space to contain the changed resource. If the modified resource data can't be written to the resource fork (for example, if there's not enough room on disk), the `resChanged` attribute is not set to 1. If this is the case and you call `UpdateResFile` or `WriteResource`, the Resource Manager won't know that the resource data has been changed. Thus, the routine won't write the modified resource data to the resource fork and won't return an error. For this reason, always make sure that the `ResError` function returns the result code `noErr` after a call to `ChangedResource`.



**IMPORTANT**

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. To do so, use the Memory Manager procedures `HGetState`, `HNoPurge`, and `HSetState` to make sure the resource data remains in memory while you change it and until the resource data is written to disk. (You can't use the `SetResAttrs` procedure for this purpose, because the changes don't take effect immediately.) First call `HGetState` and `HNoPurge`, then change the resource as necessary. To make a change to a resource permanent, use `ChangedResource` and `UpdateResFile` or `WriteResource`; then call `HSetState` when you're finished. Or, instead of calling `WriteResource` to write the resource data immediately, you can call `SetResPurge` with the `install` parameter set to `TRUE` before making changes to purgeable resource data.

If your application doesn't make its resources purgeable, or if the changes you are making to a purgeable resource don't involve routines that may cause the resource to be purged, you don't need to take these precautions. ▲

**SPECIAL CONSIDERATIONS**

The `ChangedResource` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

When called, `ChangedResource` reserves disk space for the changed resource. The procedure reserves space every time you call it, but the resource is not actually written until you call `WriteResource` or `UpdateResFile`. Thus, if you call `ChangedResource` several times before the resource is actually written, the procedure reserves much more space than is needed. If the resource is large, you may unexpectedly run out of disk space. 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.

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

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>resAttrErr</code>	-198	Attribute inconsistent with operation

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For examples of the use of the `ChangedResource` procedure, see Listing 1-2 on page 1-21 and Listing 1-11 on page 1-38.

For descriptions of the `UpdateResFile` and `WriteResource` procedures, see page 1-92 and page 1-93, respectively. For descriptions of the `SetResInfo`, `SetResAttrs`, and `SetResPurge` procedures, see page 1-82, page 1-85, and page 1-94, respectively.

For information about using the Memory Manager procedures `HGetState`, `HNoPurge`, and `HSetState`, see *Inside Macintosh: Memory*.

## AddResource

---

You can use the `AddResource` procedure to add a resource to the current resource file.

```
PROCEDURE AddResource (theData: Handle; theType: ResType;
                      theID: Integer; name: Str255);
```

<code>theData</code>	A handle to data in memory to be added as a resource to the current resource file (not a handle to an existing resource).
<code>theType</code>	The resource type of the resource to be added.
<code>theID</code>	The resource ID of the resource to be added.
<code>name</code>	The name of the resource to be added.

### DESCRIPTION

Given a handle to any type of data in memory (but not a handle to an existing resource), `AddResource` adds the given handle, resource type, resource ID, and resource name to the current resource file's resource map in memory. The `AddResource` procedure sets the `resChanged` attribute to 1; it does not set any of the resource's other attributes—that is, all other attributes are set to 0.

#### ▲ WARNING

The `AddResource` procedure doesn't verify whether the resource ID you pass in the parameter `theID` is already assigned to another resource of the same type. You should call the `UniqueID` or `Unique1ID` function to get a unique resource ID before adding a resource with `AddResource`. ▲

If the `resChanged` attribute of a resource has been set and your application calls `UpdateResFile` or quits, the Resource Manager writes both the resource map and the resource data for that resource to the resource fork of the corresponding file on disk. If the `resChanged` attribute for a resource has been set and your application calls `WriteResource`, the Resource Manager writes only the resource data for that resource to disk.

If you add a resource to the current resource file, the Resource Manager writes the entire resource map to disk when it updates the file. If you want any of your changes to the resource map or resource data to be temporary, you must restore the original information before the Resource Manager updates the file on disk.

If the value of the parameter `theData` is an empty handle (that is, a handle whose master pointer is set to `NIL`), the Resource Manager writes zero-length resource data to disk when it updates the resource. If the value of `theData` is either `NIL` or a handle to an existing resource, `AddResource` does nothing, and the `ResError` function returns the result code `addResFailed`. If the resource map becomes too large to fit in memory, `AddResource` does nothing, and `ResError` returns an appropriate result code. The same is true if the resource data in memory can't be written to the resource fork (for example, because the disk is full).

When your application calls `AddResource`, the Resource Manager attempts to reserve disk space for the new resource. If the new resource data can't be written to the resource fork (for example, if there's not enough room on disk), the `resChanged` attribute is not set to 1. If this is the case and you call `UpdateResFile` or `WriteResource`, the Resource Manager won't know that resource data has been added. Thus, the routine won't write the new resource data to the resource fork and won't return an error. For this reason, always make sure that the `ResError` function returns the result code `noErr` after a call to `AddResource`.

To copy an existing resource, get a handle to the resource you want to copy, call the `DetachResource` procedure, then call `AddResource`. To add the same resource data to several different resource forks, call the Memory Manager function `HandToHand` to duplicate the handle for each resource.

#### RESULT CODES

<code>noErr</code>	0	No error
<code>addResFailed</code>	-194	<code>AddResource</code> procedure failed

#### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For examples of the use of the `AddResource` procedure, see Listing 1-4 on page 1-24 and Listing 1-11 on page 1-38.

For descriptions of the `UpdateResFile` and `WriteResource` procedures, see page 1-92 and page 1-93, respectively. For descriptions of the `UniqueID` and `Unique1ID` functions, see page 1-96. For a description of the `DetachResource` procedure, see page 1-108.

For information about using the Memory Manager procedure `HandToHand`, see *Inside Macintosh: Memory*.

## Writing to Resource Forks

---

The Resource Manager provides three procedures that you can use to write resource information to disk. The `UpdateResFile` procedure updates the resource map and resource data of a resource fork on disk so that it matches the corresponding resource map and resource data in memory. The `WriteResource` procedure updates the resource data of just one resource on disk. The `SetResPurge` procedure sets up the Resource Manager's own purge-warning procedure so that the Memory Manager checks with the Resource Manager before purging a purgeable resource.

## UpdateResFile

---

You can use the `UpdateResFile` procedure to update the resource map and resource data for a resource fork without closing it.

```
PROCEDURE UpdateResFile (refNum: Integer);
```

`refNum`      A file reference number for a resource fork.

### DESCRIPTION

Given the reference number of a file whose resource fork is open, `UpdateResFile` performs three tasks. The first task is to change, add, or remove resource data in the file's resource fork to match the resource map in memory. Changed resource data for each resource is written only if that resource's `resChanged` bit has been set by a successful call to `ChangedResource` or `AddResource`. The `UpdateResFile` procedure calls the `WriteResource` procedure to write changed or added resources to the resource fork.

The second task is to compact the resource fork, closing up any empty space created when a resource was removed, made smaller, or made larger. If a resource is made larger, the Resource Manager writes it at the end of the resource fork rather than at its original location. It then compacts the space occupied by the original resource data. The actual size of the resource fork is adjusted when a resource is removed or made larger, but not when a resource is made smaller.

The third task is to write the resource map in memory to the resource fork if your application has called the `ChangedResource` procedure for any resource listed in the resource map or if it has added or removed a resource. All changes to resource information in the resource map become permanent at this time; if you want any of these changes to be temporary, you must restore the original information before calling `UpdateResFile`.

If there's no open resource fork with the given reference number, `UpdateResFile` does nothing, and the `ResError` function returns the result code `resNotFound`. If the value of the `refNum` parameter is 0, it represents the System file's resource fork. If you call `UpdateResFile` but the `mapReadOnly` attribute of the resource fork is set, `UpdateResFile` does nothing, and the `ResError` function returns the result code `resAttrErr`.

Because the `CloseResFile` procedure calls `UpdateResFile` before it closes the resource fork, you need to call `UpdateResFile` directly only if you want to update the file without closing it.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>resAttrErr</code>	-198	Attribute inconsistent with operation

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of `UpdateResFile`, see Listing 1-11 on page 1-38. For descriptions of the `ChangedResource`, `AddResource`, and `CloseResFile` procedures, see page 1-88, page 1-90, and page 1-110, respectively. The `WriteResource` procedure is described next.

**WriteResource**

---

You can use the `WriteResource` procedure to write resource data in memory immediately to a file's resource fork. Note that `WriteResource` does not write the resource's resource map entry to disk.

```
PROCEDURE WriteResource (theResource: Handle);
```

```
theResource
```

A handle to a resource.

**DESCRIPTION**

Given a handle to a resource, `WriteResource` checks the `resChanged` attribute of that resource. If the `resChanged` attribute is set to 1 (after a successful call to the `ChangedResource` or `AddResource` procedure), `WriteResource` writes the resource data in memory to the resource fork, then clears the `resChanged` attribute in the resource's resource map in memory.

**Note**

When your application calls `ChangedResource` or `AddResource`, the Resource Manager attempts to reserve disk space for the changed resource. If the modified resource data can't be written to the resource fork (for example, if there's not enough room on disk), the `resChanged` attribute is not set to 1. If this is the case and you call `WriteResource`, the Resource Manager won't know that the resource data has been changed. Thus, the routine won't write the modified resource data to the resource fork and won't return an error. For this reason, always make sure that the `ResError` function returns the result code `noErr` after a call to `ChangedResource` or `AddResource`. ♦

If the resource is purgeable and has been purged, `WriteResource` writes zero-length resource data to the resource fork. If the resource's `resProtected` attribute is set to 1, `WriteResource` does nothing, and the `ResError` function returns the result code `noErr`. The same is true if the `resChanged` attribute is not set (that is, set to 0). If the given handle isn't a handle to a resource, `WriteResource` does nothing, and `ResError` returns the result code `resNotFound`.

The resource fork is updated automatically when your application quits, when you call `UpdateResFile`, or when you call `CloseResFile` (which in turn calls `UpdateResFile`). Thus, you should call `WriteResource` only if you want to write just one or a few resources immediately.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For more information about resource attributes, see "The Resource Map" beginning on page 1-8. For descriptions of the `ChangedResource` and `AddResource` procedures, see page 1-88 and page 1-90, respectively. For a description of the `UpdateResFile` procedure, see page 1-92. For a description of the `CloseResFile` procedure, see page 1-110.

**SetResPurge**

---

You can use the `SetResPurge` procedure to have the Memory Manager pass the handle of a resource to the Resource Manager before purging the data specified by that handle.

```
PROCEDURE SetResPurge (install: Boolean);
```

`install`     A Boolean value that specifies whether the Memory Manager checks with the Resource Manager before purging a resource handle.

**DESCRIPTION**

Specify `TRUE` in the `install` parameter to make the Memory Manager pass the handle for a resource to the Resource Manager before purging the resource data to which the handle points. The Resource Manager determines whether the handle points to a resource in the application heap. It also checks if the resource's `resChanged` attribute is set to 1. If these two conditions are met, the Resource Manager calls the `WriteResource` procedure to write the resource's resource data to the resource fork before returning control to the Memory Manager.

Specify `FALSE` in the `install` parameter to restore the normal state, so that the Memory Manager purges resource data when it needs to without calling the Resource Manager.

You can use `SetResPurge` in applications that modify purgeable resources. You should also take precautions in such applications to ensure that the resource won't be purged while you're changing it.

**SPECIAL CONSIDERATIONS**

If you call `SetResPurge` with the `install` parameter set to `TRUE` and then call the Memory Manager procedure `MoveHHi` to move a handle to a resource, the Resource Manager calls the `WriteResource` procedure to write the resource data to disk even if the data has not been changed. To prevent this, call `SetResPurge` with the `install` parameter set to `FALSE` before you call `MoveHHi`, then call `SetResPurge` with the `install` parameter set to `TRUE` immediately after you call `MoveHHi`.

Whenever you call `SetResPurge` with the `install` parameter set to `TRUE`, the Resource Manager installs its own purge-warning procedure, overriding any purge-warning procedure you've specified to the Memory Manager.

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For descriptions of the `SetResAttrs` and `WriteResource` procedures, see page 1-85 and page 1-93, respectively.

For more information about the Memory Manager procedure `MoveHHi`, see *Inside Macintosh: Memory*.

## Getting a Unique Resource ID

---

The Resource Manager provides two routines that return a unique resource ID. The `UniqueID` function returns a resource ID that isn't currently assigned to any resource of the specified type in any open resource fork. The `Unique1ID` function returns a resource ID that isn't currently assigned to any resource of the specified type in the resource fork of the current resource file.

## UniqueID

---

You can use the `UniqueID` function to get a unique resource ID for a resource.

```
FUNCTION UniqueID (theType: ResType): Integer;
```

`theType`     A resource type.

### DESCRIPTION

The `UniqueID` function returns as its function result a resource ID greater than 0 that isn't currently assigned to any resource of the specified type in any open resource fork. You should use this function before adding a new resource to ensure that you don't duplicate a resource ID and override an existing resource.

### SPECIAL CONSIDERATIONS

In versions of system software earlier than System 7, the `UniqueID` function may return a resource ID in the range 0 through 127, which is generally reserved for system resources. You should check that the resource ID returned is not in this range. If it is, call `UniqueID` again, and continue doing so until you get a resource ID greater than 127.

In System 7 and later versions, `UniqueID` won't return a resource ID of less than 128.

### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about restrictions on resource IDs for specific resource types, see "Resource IDs" on page 1-46.

## Unique1ID

---

You can use the `Unique1ID` function to get a resource ID that's unique with respect to resources in the current resource file.

```
FUNCTION Unique1ID (theType: ResType): Integer;
```

`theType`     A resource type.



**DESCRIPTION**

The `UniqueID` function returns as its function result a resource ID greater than 0 that isn't currently assigned to any resource of the specified type in the current resource file. You should use this routine before adding a new resource to ensure that you don't duplicate a resource ID and override an existing resource.

**SPECIAL CONSIDERATIONS**

In versions of system software earlier than System 7, the `UniqueID` function may return a resource ID in the range 0 through 127, which is generally reserved for system resources. You should check that the resource ID returned is not in this range. If it is, call `UniqueID` again, and continue doing so until you get a resource ID greater than 127.

In System 7 and later versions, `UniqueID` won't return a resource ID of less than 128.

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For more information about restrictions on resource IDs for specific resource types, see "Resource IDs" on page 1-46.

## Counting and Listing Resource Types

---

The Resource Manager provides several routines that count or list resource types. The `CountResources` function returns the total number of resources of a given type that are currently available in all resource forks open to your application, and the `Count1Resources` function returns the total number of resources of a given type in the current resource file.

You can call the `GetIndResource` function repeatedly to generate handles to all resources of a given type in all resource forks open to your application. You can call the `Get1IndResource` function repeatedly to generate handles to all resources of a given type in the current resource file.

The `CountTypes` function tells you the number of resource types in all resource forks open to your application. The `Count1Types` function tells you the number of resource types in the current resource file. You can call the `GetIndType` procedure repeatedly to get all the resource types available in all resource forks open to your application. Similarly, you can call the `Get1IndType` procedure repeatedly to get all the resource types available in the current resource file.

## CountResources

---

You can use the `CountResources` function to get the total number of available resources of a given type.

```
FUNCTION CountResources (theType: ResType): Integer;
```

`theType`     A resource type.

### DESCRIPTION

Given a resource type, the `CountResources` function reads the resource maps in memory for all resource forks open to your application. It returns as its function result the total number of resources of the given type in all resource forks open to your application.

### RESULT CODE

`noErr`     0     No error

### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

## Count1Resources

---

You can use the `Count1Resources` function to get the total number of resources of a given type in the current resource file.

```
FUNCTION Count1Resources (theType: ResType): Integer;
```

`theType`     A resource type.

### DESCRIPTION

Given a resource type, the `Count1Resources` function reads the resource map in memory of the current resource file. It returns as its function result the total number of resources of the given type in the current resource file only.

### RESULT CODE

`noErr`     0     No error

## SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `Count1Resources` function, see Listing 1-10 on page 1-34.

## GetIndResource

---

You can use the `GetIndResource` function repeatedly to get handles to all resources of a given type in all resource forks open to your application.

```
FUNCTION GetIndResource (theType: ResType;
                        index: Integer): Handle;
```

`theType`     A resource type.

`index`       An integer ranging from 1 to the number of resources of a given type returned by `CountResources`, which is the number of resource types in all open resource forks.

## DESCRIPTION

Given an index ranging from 1 to the number of resources of a given type returned by `CountResources` (that is, the number of resources of that type in all resource forks open to your application), the `GetIndResource` function returns a handle to a resource of the given type. If you call `GetIndResource` repeatedly over the entire range of the index, it returns handles to all resources of the given type in all resource forks open to your application.

The function reads the resource data into memory if it's not already there, unless you've called `SetResLoad` with the `load` parameter set to `FALSE`.

**IMPORTANT**

If you've called `SetResLoad` with the `load` parameter set to `FALSE` and the data isn't already in memory, `GetIndResource` returns an empty handle (a handle whose master pointer is set to `NIL`). This can also happen if you read resource data for a purgeable resource into memory and then call `SetResLoad` with the `load` parameter set to `FALSE`. If the resource data is later purged and you call the `GetIndResource` function, `GetIndResource` returns an empty handle. You should test for an empty handle in these situations. To make the handle a valid handle to resource data in memory, you can call the `LoadResource` procedure. ♦

The `GetIndResource` function returns handles for all resources in the most recently opened resource fork first, and then for those in resource forks opened earlier in reverse chronological order.

## CHAPTER 1

### Resource Manager

#### Note

The `UseResFile` procedure affects which file the Resource Manager searches first when looking for a particular resource; this is not the case when you use `GetIndResource` to get an indexed resource. ♦

If you want to find out how many resources of a given type are in a particular resource fork, set the current resource file to that resource fork, then call `Count1Resources` and use `Get1IndResource` to get handles to the resources of that type.

If you provide an index to `GetIndResource` that's either 0 or negative, `GetIndResource` returns `NIL`, and the `ResError` function returns the result code `resNotFound`. If the given index is larger than the value returned by `CountResources`, `GetIndResource` returns `NIL`, and `ResError` returns the result code `resNotFound`. If the resource to be read won't fit into memory, `GetIndResource` returns `NIL`, and `ResError` returns the appropriate result code.

#### SPECIAL CONSIDERATIONS

The `GetIndResource` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

#### RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

#### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `CountResources` function, see page 1-98. For a description of the `UseResFile` procedure, see page 1-69. For descriptions of the `SetResLoad` and `LoadResource` procedures, see page 1-79 and page 1-80, respectively.

## Get1IndResource

---

You can use the `Get1IndResource` function repeatedly to get handles to all resources of a given type in the current resource file.

```
FUNCTION Get1IndResource (theType: ResType;  
                        index: Integer): Handle;
```

`theType`     A resource type.

`index`       An integer ranging from 1 to the number of resources of a given type returned by `Count1Resources`, which is the number of resource types in the current resource file.

**DESCRIPTION**

Given an index ranging from 1 to the number of resources of a given type returned by `Count1Resources` (that is, the number of resources of that type in the current resource file), the `Get1IndResource` function returns a handle to a resource of the given type. If you call `Get1IndResource` repeatedly over the entire range of the index, it returns handles to all resources of the given type in the current resource file.

The function reads the resource data into memory if it's not already there, unless you've called `SetResLoad` with the `load` parameter set to `FALSE`.

**IMPORTANT**

If you've called `SetResLoad` with the `load` parameter set to `FALSE` and the data isn't already in memory, `Get1IndResource` returns an empty handle (that is, a handle whose master pointer is set to `NIL`). This can also happen if you read resource data for a purgeable resource into memory and then call `SetResLoad` with the `load` parameter set to `FALSE`. If the resource data is later purged and you call the `Get1IndResource` function, `Get1IndResource` returns an empty handle. You should test for an empty handle in these situations. To make the handle a valid handle to resource data in memory, you can call the `LoadResource` procedure. ♦

If you provide an index to `Get1IndResource` that's either 0 or negative, `Get1IndResource` returns `NIL`, and the `ResError` function returns the result code `resNotFound`. If the given index is larger than the value returned by `Count1Resources`, `Get1IndResource` returns `NIL`, and `ResError` returns the result code `resNotFound`. If the resource to be read won't fit into memory, `Get1IndResource` returns `NIL`, and `ResError` returns the appropriate result code.

**SPECIAL CONSIDERATIONS**

The `Get1IndResource` function may move or purge memory blocks in the application heap. Your application should not call this function at interrupt time.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `Get1IndResource` function, see Listing 1-10 on page 1-34.

For a description of the `Count1Resources` function, see page 1-98. For a description of the `UseResFile` procedure, see page 1-69. For descriptions of the `SetResLoad` and `LoadResource` procedures, see page 1-79 and page 1-80, respectively.

## CountTypes

---

You can use the `CountTypes` function to get the number of resource types in all resource forks open to your application.

```
FUNCTION CountTypes: Integer;
```

### DESCRIPTION

The `CountTypes` function reads the resource maps in memory for all resource forks open to your application. It returns an integer representing the total number of unique resource types.

### RESULT CODE

`noErr`    0    No error

### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

## Count1Types

---

You can use the `Count1Types` function to get the number of resource types in the current resource file.

```
FUNCTION Count1Types: Integer;
```

### DESCRIPTION

The `Count1Types` function reads the resource map in memory for the current resource file. It returns an integer representing the total number of unique resource types in the current resource file.

### RESULT CODE

`noErr`    0    No error

### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

## GetIndType

---

You can call the `GetIndType` procedure repeatedly to get all the resource types available in all resource forks open to your application.

```
PROCEDURE GetIndType (VAR theType: ResType; index: Integer);
```

`theType`     `GetIndType` returns, in this parameter, the resource type for the specified index among all the resource forks open to your application.

`index`        An integer ranging from 1 to the number of resource types in all resource forks open to your application.

### DESCRIPTION

Given an index number from 1 to the number of resource types in all resource forks open to your application (as returned by `CountTypes`), the `GetIndType` procedure returns a resource type in the parameter `theType`. You can call `GetIndType` repeatedly over the entire range of the index to get all the resource types available in all resource forks open to your application. If the given index isn't in the range from 1 to the number of resource types as returned by `CountTypes`, `GetIndType` returns four null characters (ASCII code 0).

### SPECIAL CONSIDERATIONS

The `GetIndType` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

### RESULT CODE

`noErr`     0     No error

### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about the `CountTypes` function, see page 1-102.

## Get1IndType

---

You can use the `Get1IndType` procedure to get all the resource types available in the current resource file.

```
PROCEDURE Get1IndType (VAR theType: ResType; index: Integer);
```

`theType`      `Get1IndType` returns, in this parameter, the resource type with the specified index in the current resource file.

`index`        An integer ranging from 1 to the number of resource types in the current resource file.

### DESCRIPTION

Given an index number from 1 to the number of resource types in the current resource file (as returned by `Count1Types`), the `Get1IndType` procedure returns a resource type in the parameter `theType`. You can call `Get1IndType` repeatedly over the entire range of the index to get all the resource types available in the current resource file. If the given index isn't in the range from 1 to the number of resource types as returned by `Count1Types`, `Get1IndType` returns four null characters (ASCII code 0).

### SPECIAL CONSIDERATIONS

The `Get1IndType` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

### RESULT CODE

`noErr`      0      No error

### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `Count1Types` function, see page 1-102.

## Getting Resource Sizes

---

The Resource Manager provides two routines that allow you to get the size of a resource. The `GetResourceSizeOnDisk` and `GetMaxResourceSize` functions get the exact size and maximum size, respectively, of a resource. To change the size of a resource on disk, use the `SetResourceSize` procedure.



## GetResourceSizeOnDisk

---

You can use the `GetResourceSizeOnDisk` function to get the exact size of a resource. The `GetResourceSizeOnDisk` function is also available as the `SizeResource` function.

```
FUNCTION GetResourceSizeOnDisk (theResource: Handle): LongInt;
```

`theResource`

A handle to a resource.

### DESCRIPTION

Given a handle to a resource, the `GetResourceSizeOnDisk` function checks the resource on disk (not in memory) and returns its exact size, in bytes. If the handle isn't a handle to a valid resource, `GetResourceSizeOnDisk` returns `-1`, and `ResError` returns the result code `resNotFound`.

You can call `GetResourceSizeOnDisk` before reading a resource into memory to make sure there's enough memory available to do so successfully.

### RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

## GetMaxResourceSize

---

You can use the `GetMaxResourceSize` function to get the approximate size of a resource. The `GetMaxResourceSize` function is also available as the `MaxSizeRsrc` function.

```
FUNCTION GetMaxResourceSize (theResource: Handle): LongInt;
```

`theResource`

Handle to a resource.

**DESCRIPTION**

Like `GetResourceSizeOnDisk`, `GetMaxResourceSize` takes a handle and returns the size of the corresponding resource. However, `GetMaxResourceSize` does not check the resource on disk; instead, it either checks the resource size in memory or, if the resource is not in memory, calculates its size, in bytes, on the basis of information in the resource map in memory. This gives you an approximate size for the resource that you can count on as the resource's maximum size. It's possible that the resource is actually smaller than the offsets in the resource map indicate because the file has not yet been compacted. If you want the exact size of a resource on disk, either call `GetResourceSizeOnDisk` or call `UpdateResFile` before calling `GetMaxResourceSize`.

If the value of the `theResource` parameter isn't a handle to a valid resource, `GetMaxResourceSize` returns `-1`, and `ResError` returns the result code `resNotFound`.

**RESULT CODES**

<code>noErr</code>	<code>0</code>	No error
<code>resNotFound</code>	<code>-192</code>	Resource not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For a description of the `UpdateResFile` and `GetResourceSizeOnDisk` routines, see page 1-92 and page 1-105, respectively.

## Disposing of Resources

---

The Resource Manager provides three procedures for disposing of resources. The `ReleaseResource` procedure releases the memory associated with a resource, setting the handle's master pointer to `NIL`, thus making your application's handle to the resource invalid. The `DetachResource` procedure sets a resource's handle in the resource map to `NIL` but keeps the resource data in memory. The `RemoveResource` procedure removes the resource's entry from the resource map in memory; the Resource Manager removes the resource data from memory (and from the file's resource fork) when it updates the file's resource fork.

## ReleaseResource

---

You can use the `ReleaseResource` procedure to release the memory a resource occupies when you have finished using it.

```
PROCEDURE ReleaseResource (theResource: Handle);
```

```
theResource
```

A handle to a resource.

### DESCRIPTION

Given a handle to a resource, `ReleaseResource` releases the memory occupied by the resource data, if any, and sets the master pointer of the resource's handle in the resource map in memory to `NIL`. If your application previously obtained a handle to that resource, the handle is no longer valid. If your application subsequently calls the Resource Manager to get the released resource, the Resource Manager assigns a new handle.

If the given resource isn't a handle to a resource, `ReleaseResource` does nothing, and `ResError` returns the result code `resNotFound`. Be aware that `ReleaseResource` won't release a resource whose `resChanged` attribute has been set, but `ResError` still returns the result code `noErr`.

### SPECIAL CONSIDERATIONS

The `ReleaseResource` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

### RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about releasing resources, see "Releasing and Detaching Resources" beginning on page 1-22. For an example of the use of the `ReleaseResource` procedure, see Listing 1-8 on page 1-29.

## DetachResource

---

You can use the `DetachResource` procedure to set the value of a resource's handle in the resource map in memory to `NIL` while keeping the resource data in memory.

```
PROCEDURE DetachResource (theResource: Handle);
```

```
theResource
```

A handle to a resource.

### DESCRIPTION

Given a handle to a resource, `DetachResource` sets the value of the resource's handle in the resource map in memory to `NIL`. After this call, the Resource Manager no longer recognizes the handle as a handle to a resource. However, `DetachResource` does not release the memory used for the resource data, and the master pointer is still valid. Thus, you can access the resource data directly by using the handle.

If your application subsequently calls a Resource Manager routine to get the released resource, the Resource Manager assigns a new handle. If the parameter `theResource` doesn't contain a handle to a resource or if the resource's `resChanged` attribute is set, `DetachResource` does nothing. To determine whether either of these errors occurred, call `ResError`.

You can use `DetachResource` if you want to access the resource data directly without using Resource Manager routines. You can also use the `DetachResource` procedure to keep resource data in memory after closing a resource fork.

To copy a resource and install an entry for the duplicate in the resource map, call `DetachResource`, then call `AddResource` using a different resource ID.

### RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found
<code>resAttrErr</code>	-198	Attribute does not permit operation

### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For more information about detaching resources, see "Releasing and Detaching Resources" beginning on page 1-22. For an example of the use of the `DetachResource` procedure, see Listing 1-4 on page 1-24.

For a description of the `AddResource` procedure, see page 1-90.

## RemoveResource

---

You can use the `RemoveResource` procedure to remove a resource's entry from the current resource file's resource map in memory. The `RemoveResource` procedure is also available as the `RmveResource` procedure.

```
PROCEDURE RemoveResource (theResource: Handle);
```

`theResource`

A handle to a resource.

### DESCRIPTION

Given a handle to a resource in the current resource file, `RemoveResource` removes the resource entry (resource type, resource ID, resource name, if any, and resource attributes) from the current resource file's resource map in memory.

The `RemoveResource` procedure doesn't immediately release the memory occupied by the resource data; instead, the Resource Manager releases the memory when your application quits, when you call `UpdateResFile`, or when you call `CloseResFile` (which in turn calls `UpdateResFile`). If the `resProtected` attribute for the resource is set or if the `theResource` parameter doesn't contain a handle to a resource, `RemoveResource` does nothing, and `ResError` returns the result code `rmvResFailed`.

### IMPORTANT

If you've removed a resource, the Resource Manager writes the entire resource map when it updates the resource fork, and all changes made to the resource map become permanent. If you want any of the changes to be temporary, you should restore the original information before the Resource Manager updates the resource fork. ▲

If you want to release the memory before updating or closing the resource fork, call the Memory Manager procedure `DisposeHandle` after you call `RemoveResource`.

### SPECIAL CONSIDERATIONS

The `RemoveResource` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

### RESULT CODES

<code>noErr</code>	0	No error
<code>rmvResFailed</code>	-196	<code>RemoveResource</code> procedure failed

SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For descriptions of the `AddResource` and `UpdateResFile` procedures, see page 1-90 and page 1-92, respectively. The `CloseResFile` procedure is described next.

For more information about the Memory Manager procedure `DisposeHandle`, see *Inside Macintosh: Memory*.

## Closing Resource Forks

---

When your application terminates, the Resource Manager automatically closes every resource fork open to your application except the System file's resource fork. The `CloseResFile` procedure allows you to close a resource fork before your application terminates.

## CloseResFile

---

You can use the `CloseResFile` procedure to close a resource fork before your application terminates.

```
PROCEDURE CloseResFile (refNum: Integer);
```

`refNum`      The file reference number for the resource fork to close.

DESCRIPTION

Given a file reference number for a file whose resource fork is open, the `CloseResFile` procedure performs four tasks. First, it updates the file by calling the `UpdateResFile` procedure. Second, it releases the memory occupied by each resource in the resource fork by calling the `DisposeHandle` procedure. Third, it releases the memory occupied by the resource map. The fourth task is to close the resource fork.

If the `refNum` parameter does not contain a file reference number for a file whose resource fork is open, `CloseResFile` does nothing, and the `ResError` function returns the result code `resFNotFound`. If the value of the `refNum` parameter is 0, it represents the System file and is ignored. You cannot close the System file's resource fork.

When your application terminates, the Resource Manager automatically closes every resource fork open to your application except the System file's resource fork. You need to call the `CloseResFile` procedure only if you want to close a resource fork before your application terminates.

RESULT CODES

<code>noErr</code>	0	No error
<code>resFNotFound</code>	-193	Resource file not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `CloseResFile` procedure, see Listing 1-9 on page 1-32.

For descriptions of the `UpdateResFile` and `ReleaseResource` procedures, see page 1-92 and page 1-107, respectively.

## Reading and Writing Partial Resources

---

You can use the `ReadPartialResource`, `WritePartialResource`, and `SetResourceSize` procedures to work with a portion of a large resource that may not otherwise fit in memory.

When using partial resource routines, you should call the `SetResLoad` procedure, specifying `FALSE` for the load parameter, before you call `GetResource`. Using the `SetResLoad` procedure prevents the Resource Manager from reading the entire resource into memory. Be sure to restore the normal state by calling `SetResLoad` again, with the load parameter set to `TRUE`, immediately after you call `GetResource`. Then use `ReadPartialResource` to read a portion of the resource into a buffer and `WritePartialResource` as needed to write a portion of the resource from a buffer to disk.

Note that the partial resources routines work with the data in the memory pointed to by the `buffer` parameter, not the memory referenced through the resource's handle. Therefore, you may experience problems if you have a copy of a resource in memory when you are using the partial resource routines. If you have modified the copy in memory and then access the resource on disk using the `ReadPartialResource` procedure, `ReadPartialResource` reads the data on disk, not the data in memory, which is referenced through the resource's handle. Similarly, `WritePartialResource` writes data from the specified buffer, not from the data in memory, which is referenced through the resource's handle.

## ReadPartialResource

---

You can use the `ReadPartialResource` procedure to read part of a resource into memory and work with a small subsection of a large resource.

```
PROCEDURE ReadPartialResource (theResource: Handle;
                               offset: LongInt; buffer: UNIV Ptr;
                               count: LongInt);
```

`theResource`

A handle to a resource.

`offset`

The beginning of the resource subsection to be read, measured in bytes from the beginning of the resource.

## CHAPTER 1

### Resource Manager

`buffer`      A pointer to the buffer into which the partial resource is to be read.  
`count`        The length of the resource subsection.

#### DESCRIPTION

The `ReadPartialResource` procedure reads the resource subsection identified by the `theResource`, `offset`, and `count` parameters into a buffer specified by the `buffer` parameter. Your application is responsible for the buffer's memory management. You cannot use the `ReleaseResource` procedure to release the memory the buffer occupies.

The `ReadPartialResource` procedure always tries to read resources from disk. If a resource is already in memory, the Resource Manager still reads it from disk, and the `ResError` function returns the result code `resourceInMemory`. If you try to read past the end of a resource or the value of the `offset` parameter is out of bounds, `ResError` returns the result code `inputOutOfBounds`. If the handle in the parameter `theResource` doesn't refer to a resource in an open resource fork, `ResError` returns the result code `resNotFound`.

When using partial resource routines, you should call the `SetResLoad` procedure, specifying `FALSE` for the `load` parameter, before you call `GetResource`. Using the `SetResLoad` procedure prevents the Resource Manager from reading the entire resource into memory. Be sure to restore the normal state by calling `SetResLoad` again, with the `load` parameter set to `TRUE`, immediately after you call `GetResource`. Then use `ReadPartialResource` to read a portion of the resource into a buffer.

#### Note

If the entire resource is in memory and you want only part of its data, it's faster to use the Memory Manager procedure `BlockMove` instead of the `ReadPartialResource` procedure. If you read a partial resource into memory and then change its size, you can use `SetResourceSize` to change the entire resource's size on disk as necessary. ♦

#### SPECIAL CONSIDERATIONS

The `ReadPartialResource` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

#### ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `ReadPartialResource` are

Trap macro	Selector
<code>_ResourceDispatch</code>	<code>\$7001</code>



## RESULT CODES

noErr	0	No error
resourceInMemory	-188	Resource already in memory
inputOutOfBounds	-190	Offset or count out of bounds
resNotFound	-192	Resource not found

## SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For an example of the use of the `ReadPartialResource` procedure, see Listing 1-12 on page 1-41.

For descriptions of the `GetResource`, `SetResLoad`, and `ReleaseResource` routines, see page 1-73, page 1-79, and page 1-107, respectively. For a description of the `SetResourceSize` procedure, see page 1-115.

For information about the Memory Manager procedure `BlockMove`, see *Inside Macintosh: Memory*.

## WritePartialResource

---

You can use the `WritePartialResource` procedure to write part of a resource to disk when working with a small subsection of a large resource.

```
PROCEDURE WritePartialResource (theResource: Handle;
                               offset: LongInt; buffer: UNIV Ptr;
                               count: LongInt);
```

`theResource`

A handle to a resource.

`offset`

The beginning of the resource subsection to write, measured in bytes from the beginning of the resource.

`buffer`

A pointer to the buffer containing the data to write.

`count`

The length of the resource subsection to write.

## DESCRIPTION

The `WritePartialResource` procedure writes the data specified by the `buffer` parameter to the resource subsection identified by the `theResource`, `offset`, and `count` parameters. Your application is responsible for the buffer's memory management.

## CHAPTER 1

### Resource Manager

If the disk or the file is locked, the `ResError` function returns an appropriate File Manager result code. If you try to write past the end of a resource, the Resource Manager attempts to enlarge the resource. The `ResError` function returns the result code `writingPastEnd` if the attempt succeeds. If the Resource Manager cannot enlarge the resource, `ResError` returns an appropriate File Manager result code. If you pass an invalid value in the `offset` parameter, `ResError` returns the result code `inputOutOfBounds`.

The `WritePartialResource` procedure tries to write the data from the buffer to disk. If the attempt is successful and the resource data (referenced through the resource's handle) is in memory, `ResError` returns the result code `resourceInMemory`. In this situation, be aware that the data of the resource subsection on disk matches the data from the buffer, not the resource data referenced through the resource's handle. If the attempt to write the data from the buffer to the disk fails, `ResError` returns an appropriate error.

When using partial resource routines, you should call the `SetResLoad` procedure, specifying `FALSE` for the `load` parameter, before you call `GetResource`. Doing so prevents the Resource Manager from reading the entire resource into memory. Be sure to restore the normal state by calling `SetResLoad` again, with the `load` parameter set to `TRUE`, immediately after you call `GetResource`.

If you read a partial resource into memory and then change its size, you must use `SetResourceSize` to change the entire resource's size on disk as necessary before you write the partial resource.

#### SPECIAL CONSIDERATIONS

The `WritePartialResource` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

#### ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `WritePartialResource` are

Trap macro	Selector
<code>_ResourceDispatch</code>	<code>\$7002</code>

#### RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>dskFulErr</code>	<code>-34</code>	Disk full
<code>resourceInMemory</code>	<code>-188</code>	Resource already in memory
<code>writingPastEnd</code>	<code>-189</code>	Writing past end of file
<code>inputOutOfBounds</code>	<code>-190</code>	Offset or count out of bounds

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For descriptions of the `GetResource` and `SetResLoad` routines, see page 1-73 and page 1-79, respectively. The `SetResourceSize` procedure is described next.

**SetResourceSize**

---

You can use the `SetResourceSize` procedure to change the size of a resource on disk. This procedure is normally used only with `ReadPartialResource` and `WritePartialResource`.

```
PROCEDURE SetResourceSize (theResource: Handle; newSize: LongInt);
```

`theResource`

A handle to a resource.

`newSize`

The size, in bytes, that you want the resource to occupy on disk.

**DESCRIPTION**

Given a handle to a resource, `SetResourceSize` sets the size field of the specified resource on disk without writing the resource data. You can change the size of any resource, regardless of the amount of memory you have available.

If the specified size is smaller than the resource's current size on disk, you lose any data from the cutoff point to the end of the resource. If the specified size is larger than the resource's current size on disk, all data is preserved, but the additional area is uninitialized (arbitrary data).

If you read a partial resource into memory and then change its size, you must use `SetResourceSize` to change the entire resource's size on disk as necessary. For example, suppose the entire resource occupies 1 MB and you use `ReadPartialResource` to read in a 200 KB portion of the resource. If you then increase the size of this partial resource to 250 KB, you must call `SetResourceSize` to set the size of the resource on disk to 1.05 MB. Note that in this case you must also keep track of the resource data on disk and move any data that follows the original partial resource on disk. Otherwise, there will be no space for the additional 50 KB when you call `WritePartialResource` to write the modified partial resource to disk.

Under certain circumstances, the Resource Manager overrides the size you set with a call to `SetResourceSize`. For instance, suppose you read an entire resource into memory by calling `GetResource` or related routines, then use `SetResourceSize` successfully to set the resource size on disk, and finally attempt to write the resource to disk using `UpdateResFile` or `WriteResource`. In this case, the Resource Manager adjusts the resource size on disk to conform with the size of the resource in memory.

## CHAPTER 1

### Resource Manager

If the disk is locked or full, or the file is locked, the `SetResourceSize` procedure does nothing, and the `ResError` function returns an appropriate File Manager result code. If the resource is in memory, the Resource Manager tries to set the size of the resource on disk. If the attempt succeeds, `ResError` returns the result code `resourceInMemory`, and the Resource Manager does not update the copy in memory. If the attempt fails, `ResError` returns an appropriate File Manager result code.

#### SPECIAL CONSIDERATIONS

The `SetResourceSize` procedure may move or purge memory blocks in the application heap. Your application should not call this procedure at interrupt time.

#### ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SetResourceSize` are

Trap macro	Selector
<code>_ResourceDispatch</code>	<code>\$7003</code>

#### RESULT CODES

<code>noErr</code>	<code>0</code>	No error
<code>resourceInMemory</code>	<code>-188</code>	Resource already in memory
<code>writingPastEnd</code>	<code>-189</code>	Writing past end of file

#### SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

## Getting and Setting Resource Fork Attributes

---

The `GetResFileAttrs` function and the `SetResFileAttrs` procedure allow you to get and set a resource fork's attributes. You usually don't need to use these routines.

### GetResFileAttrs

---

You can use the `GetResFileAttrs` function to get the attributes of a resource fork.

```
FUNCTION GetResFileAttrs (refNum: Integer): Integer;
```

`refNum`      A file reference number for the resource fork whose attributes you want to get.

**DESCRIPTION**

Given a file reference number, the `GetResFileAttrs` function returns the attributes of the file's resource fork. Specify 0 in the `refNum` parameter to get the attributes of the System file's resource fork. If there's no open resource fork for the given file reference number, `GetResFileAttrs` does nothing, and the `ResError` function returns the result code `resFNotFound`.

Like individual resources, resource forks have attributes that are specified by bits in the low-order byte of a word. The Resource Manager provides the following masks for testing these bits:

```
CONST
    mapReadOnly    = 128;  {set if file is read-only}
    mapCompact     = 64;   {set to compact file on update}
    mapChanged     = 32;   {set to write map on update}
```

When the `mapReadOnly` attribute is set to 1, the Resource Manager doesn't write anything to the resource fork on disk. It also doesn't check whether the resource data can be written to disk when the resource map is modified. When this attribute is set to 1, the `UpdateResFile` and `WriteResource` procedures do nothing, but the `ResError` function returns the result code `noErr`.

When the `mapCompact` attribute is set to 1, the Resource Manager compacts the resource fork when it updates the file. The Resource Manager sets this attribute when a resource is removed or when a resource is made larger and thus must be written at the end of a resource fork. You may want to set the `mapCompact` attribute to force the Resource Manager to compact a resource fork when your changes have made resources smaller.

When the `mapChanged` attribute is set to 1, the Resource Manager writes the resource map to disk when the file is updated. For example, you can set `mapChanged` if you've changed resource attributes only and don't want to call `ChangedResource` because you don't want to write the resource data to disk.

**SPECIAL CONSIDERATIONS**

The Resource Manager sets the `mapChanged` attribute for the resource fork when you call the `ChangedResource`, the `AddResource`, or the `RemoveResource` procedure.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resFNotFound</code>	-193	Resource file not found

## SEE ALSO

To check for errors, call the `ResError` function as described on page 1-51.

For descriptions of the `ChangedResource` and `AddResource` procedures, see page 1-88 and page 1-90, respectively. For descriptions of the `UpdateResFile` and `WriteResource` procedures, see page 1-92 and page 1-93, respectively. For a description of the `RemoveResource` procedure, see page 1-109.

## SetResFileAttrs

---

You can use the `SetResFileAttrs` procedure to change a resource fork's attributes.

```
PROCEDURE SetResFileAttrs (refNum: Integer; attrs: Integer);
```

`refNum`      A file reference number for the resource fork whose attributes you want to set.

`attrs`        The attributes to set.

## DESCRIPTION

Given a file reference number, the `SetResFileAttrs` procedure sets the attributes of the file's resource fork to those specified in the `attrs` parameter. If the `refNum` parameter is 0, it represents the System file's resource fork. However, you shouldn't change the attributes of the System file's resource fork. If there's no resource fork with the given reference number, `SetResFileAttrs` does nothing, and the `ResError` function returns the result code `noErr`.

Like individual resources, resource forks have attributes that are specified by bits in the low-order byte of a word. The Resource Manager provides the following masks for setting these bits:

```
CONST
    mapReadOnly      = 128; {set to make file read-only}
    mapCompact       = 64;  {set to compact file on update}
    mapChanged       = 32;  {set to write map on update}
```

When the `mapReadOnly` attribute is set to 1, the Resource Manager doesn't write anything to the resource fork on disk. It also doesn't check whether the resource data can be written to disk when the resource map is modified. When this attribute is set to 1, the `UpdateResFile` and `WriteResource` procedures do nothing, but the `ResError` function returns the result code `noErr`.

**▲ WARNING**

If you set the `mapReadOnly` attribute but later clear it, the resource data is written to disk even if there's no room for it. This operation may destroy the resource fork. ▲

When the `mapCompact` attribute is set to 1, the Resource Manager compacts the resource fork when it updates the file. The Resource Manager sets this attribute when a resource is removed or when a resource is made larger and thus must be written at the end of a resource fork. You may want to set the `mapCompact` attribute to force the Resource Manager to compact a resource fork when your changes make resources smaller.

When the `mapChanged` attribute is set to 1, the Resource Manager writes the resource map to disk when the file is updated. For example, you can set `mapChanged` if you've changed resource attributes only and don't want to call `ChangedResource` because you don't want to write the resource data to disk.

When the Resource Manager first creates a resource fork after a call to `FSpOpenResFile` or a related routine, it does not set any of the resource forks's attributes—that is, they are all set to 0.

**SPECIAL CONSIDERATIONS**

The Resource Manager sets the `mapChanged` attribute for the resource fork when you call the `ChangedResource`, the `AddResource`, or the `RemoveResource` procedure.

**RESULT CODES**

<code>noErr</code>	0	No error
<code>resFNotFound</code>	-193	Resource file not found

**SEE ALSO**

To check for errors, call the `ResError` function as described on page 1-51.

For descriptions of the `ChangedResource` and `AddResource` procedures, see page 1-88 and page 1-90, respectively. For descriptions of the `UpdateResFile` and `WriteResource` procedures, see page 1-92 and page 1-93, respectively. For a description of the `RemoveResource` procedure, see page 1-109.

**Accessing Resource Entries in a Resource Map**

The `RsrcMapEntry` function is an advanced routine that provides a way to access the resource entries in a resource map in memory. Because the Resource Manager provides routines for opening, retrieving, and changing resources, there's usually no reason to access resource entries directly.

## RsrcMapEntry

---

To access the resource entries in a resource map in memory directly, you can use the `RsrcMapEntry` function.

```
FUNCTION RsrcMapEntry (theResource: Handle): LongInt;
```

`theResource`

A handle to a resource.

### DESCRIPTION

Given a handle to a resource, `RsrcMapEntry` returns the offset of the specified resource's entry from the beginning of the resource map in memory. If it doesn't find the resource entry, `RsrcMapEntry` returns 0, and the `ResError` function returns the result code `resNotFound`. If you pass a handle whose value is `NIL`, `RsrcMapEntry` returns arbitrary data, but `ResError` returns the result code `noErr`.

### ▲ WARNING

Because the Resource Manager provides routines for opening, retrieving, and changing resources, there's usually no reason to access a resource map directly. To avoid damaging the file for which it's called, you should use `RsrcMapEntry` extremely carefully. ▲

### RESULT CODES

<code>noErr</code>	0	No error
<code>resNotFound</code>	-192	Resource not found

### SEE ALSO

For an overview of the resource map, see "The Resource Map" beginning on page 1-8. For details of the structure of the resource map, see Figure 1-14 on page 1-123.

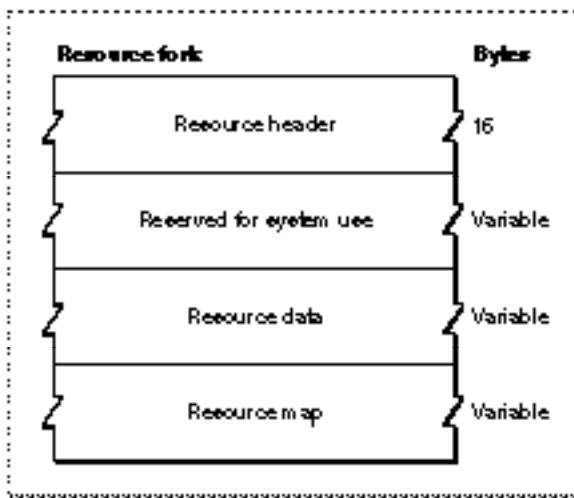


## Resource File Format

You need to know the exact format of a resource fork, which is described in this section, only if you're writing an application that creates or modifies a resource fork directly, without using Resource Manager routines.

Figure 1-11 shows the format of a compiled resource fork.

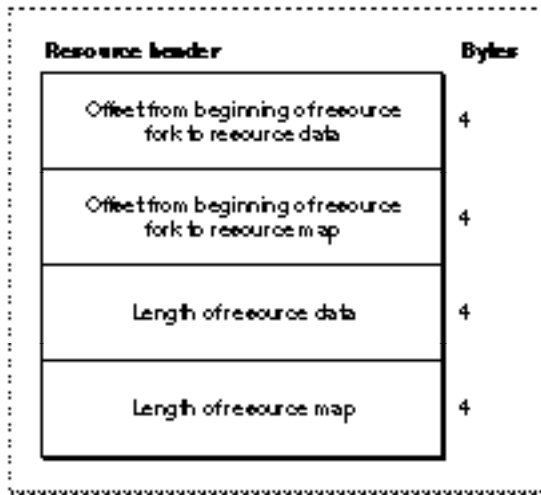
**Figure 1-11** Format of a resource fork



As Figure 1-11 shows, every resource fork begins with a resource header. Because the resource header contains an offset to the resource map, the resource map does not necessarily have to be located at the end of the resource fork.

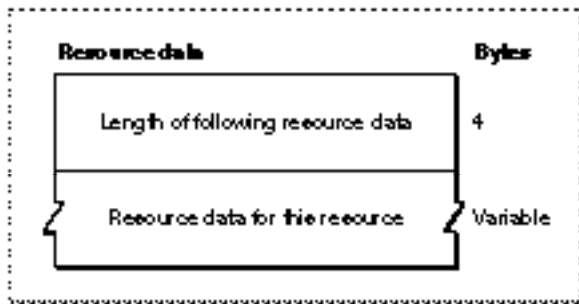
Figure 1-12 shows the format of a resource header.

**Figure 1-12** Format of a resource header in a resource fork



The resource data in a resource fork consists of the data in its individual resources. Figure 1-13 shows the format of resource data for a single resource.

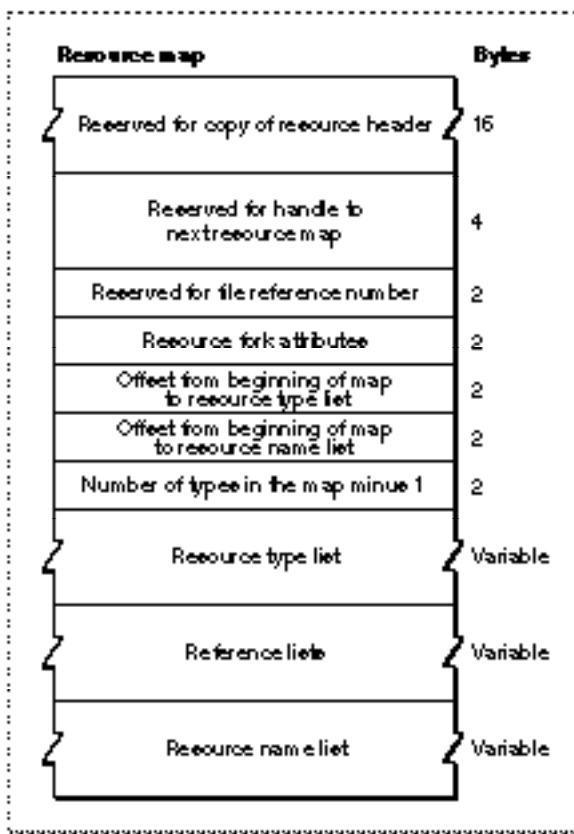
**Figure 1-13** Format of resource data for a single resource



For detailed descriptions of the resource data for various standard resource types, see the appropriate books in the *Inside Macintosh* series.

The resource data in a resource fork is followed by the resource map. Figure 1-14 shows the format of a resource map.

Figure 1-14 Format of the resource map in a resource fork

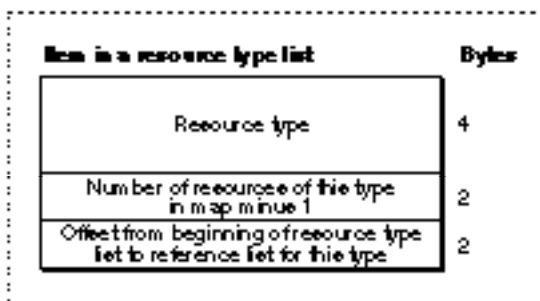


After reading the resource map into memory, the Resource Manager stores the indicated information in the reserved areas at the beginning of the map.

Each item in a resource type list specifies one resource type used in the resource fork, the number of resources of that type, and the location of the reference list for that type.

Figure 1-15 shows the format of an item in a resource type list.

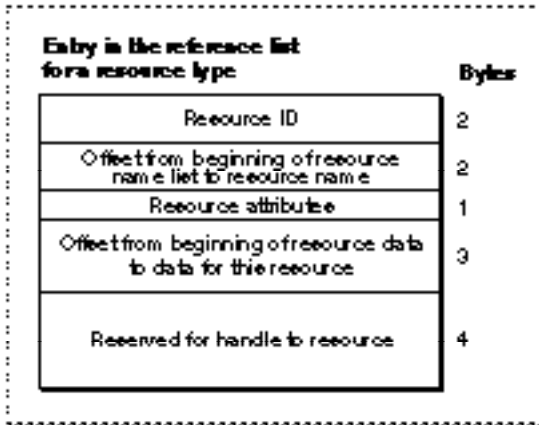
Figure 1-15 Format of an item in a resource type list



The resource type list is followed by the reference lists for each type of resource. Each resource type has a corresponding reference list that contains entries for each resource of that type. The reference lists are contiguous and in the same order as the types in the resource type list.

Figure 1-16 shows the format of an entry in a reference list.

**Figure 1-16** Format of an entry in the reference list for a resource type



If a resource does not have a name, the offset to the resource name in the resource's entry in the reference list is -1. If a resource does have a name, the offset identifies the location of the name's entry in the resource name list. Figure 1-17 shows the format of an item in the resource name list.

**Figure 1-17** Format of an item in a resource name list

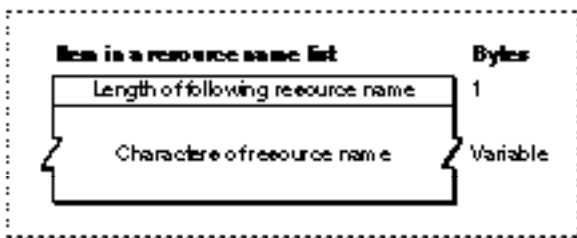
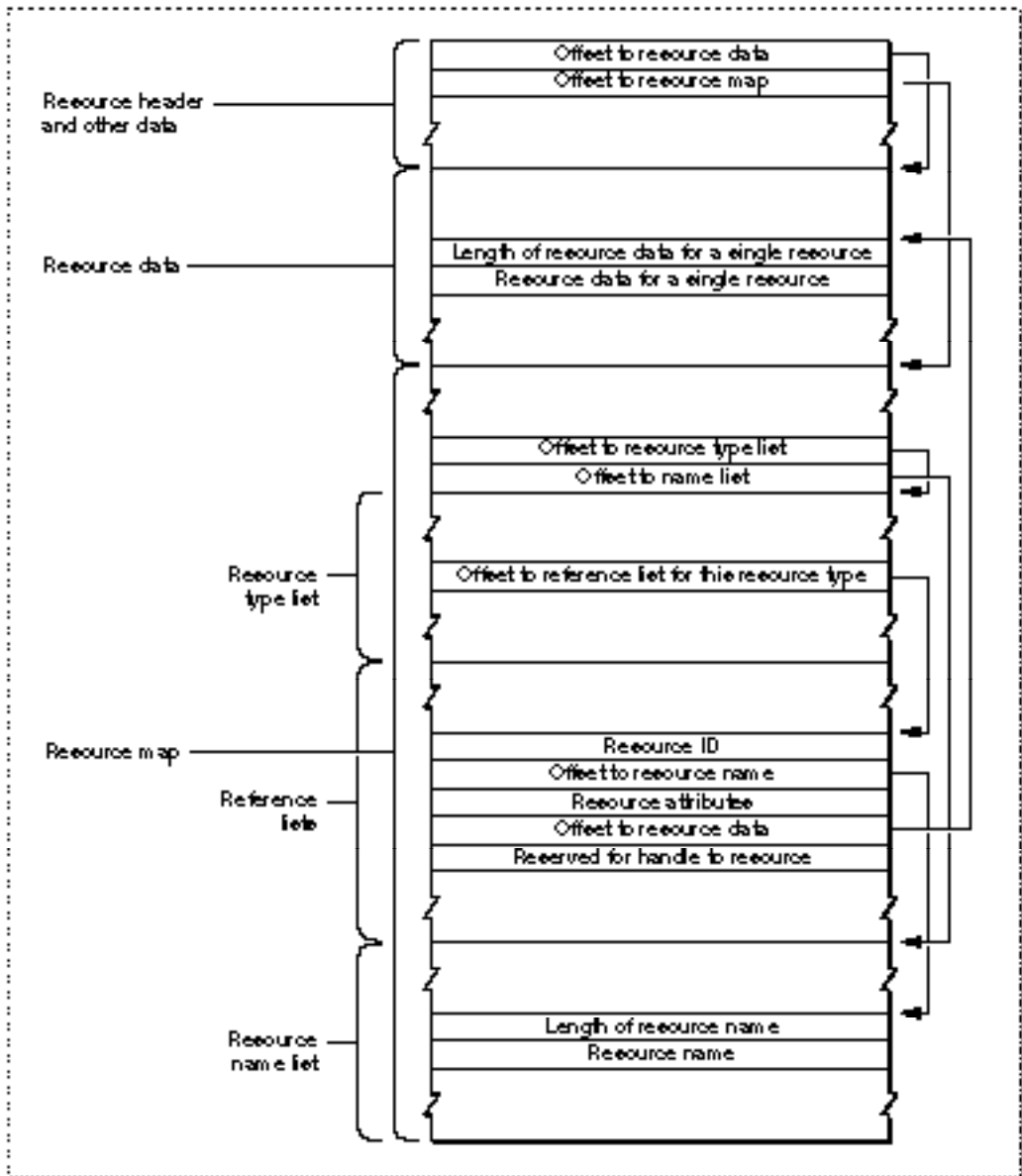


Figure 1-18 illustrates the use of various offsets in the resource header and resource map, including the offsets for an entry in a reference list for an individual resource. Although the figure shows the resource map after the resource data, the resource map can be located almost anywhere in the resource fork as long as the offset to the map in the resource header points to the right location.

**Figure 1-18** Offsets in a resource fork and an entry for a single resource in a reference list



## Resources in the System File

---

The System file's resource fork contains resources that are shared by all applications and system software. The sections that follow describe these resources.

▲ **WARNING**

Your application should not directly add resources to, delete resources from, or modify resources in the System file. ▲

If your application needs to install drivers, you should ship it with the Installer and an Installer script for drivers. To distribute the Installer, you need to license the Apple system software, which includes the Installer.

The next section describes resources in the System file that provide information about the computer on which your application is currently running, such as the user's name, the computer name, and the current printer type. You can use Resource Manager routines or the `Gestalt` function to obtain this information. Subsequent sections list system software routines kept in packages in the System file and function key resources.

In System 7 and later versions of system software, users can add resources such as scripts, keyboards, and sounds to the System file by dragging the resource icons to the System Folder. Desk accessories and resources such as system extensions are stored in the subdirectories of the System Folder, not in the System file. In System 7.0, users can also add resources such as fonts to the System file by dragging their icons to the System Folder. In System 7.1 and later versions, fonts are stored in a subdirectory of the System Folder rather than in the System file. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for details.)

The folders in the System Folder and some system resources are represented by standard icons. "Standard Icons" beginning on page 1-129 lists the most important standard icons.

## User Information Resources

---

The following resources in the resource fork of the System file provide the user's name, the computer name, the model of computer, the icon for the computer model, and the current printer type:

Information	Resource ID	Resource type	Description
User name	-16096	'STR'	The name of the person who “owns” the computer or is the current user. Use the <code>GetString</code> function with this resource ID to return the user name.
Computer name	-16413	'STR'	The name of the computer, which is distinct from the user name and from any internal hard disks that may be present. The default name of the computer is “ <i>User name's Macintosh.</i> ” Use the <code>GetString</code> function with this resource ID to return the computer name.
Computer model	-16395	'STR#'	The model of the computer, such as Macintosh SE/30 or Macintosh IIfx. The Gestalt selector for the computer model is <code>gestaltMachineModel</code> , and the Gestalt function returns a response value for this selector. You can use this value as an index into the 'STR#' resource using the <code>GetIndString</code> procedure. You should never use the model of the computer as an indication of what software features or hardware may be available.
Computer icon	Value of response parameter returned from Gestalt	'ICN#' 'icl4' 'icl8' 'ics#' 'ics4' 'ics8'	The icon for the computer model, such as the Macintosh II or Macintosh IIfx. The icons for computers are stored in icon families. The Gestalt selector for the computer icon is <code>gestaltMachineIcon</code> . Use the value from the response value for this selector as the resource ID of the icon resource you want. (For more information about icon families, see the chapter “Finder Interface” in <i>Inside Macintosh: Macintosh Toolbox Essentials</i> .)
Printer type	-8192	'STR'	The type of printer to which the computer sends documents, such as a LaserWriter printer. There is no method for retrieving the name of the printer. Use the <code>GetString</code> function to return the type of printer.

## Resource Manager

You should use `GetString`, not `GetResource`, to get the string for the user name or the computer name. Once you have the string, you should not release it, dispose of it, or make it purgeable. You will find that the resource was already loaded when you asked for it, so it should remain loaded when you are finished. Do not change the contents of either of these strings or mark them as changed. System 6 and earlier versions of system software do not necessarily have the computer name resource, and for this reason you should provide error checking as appropriate.

The `GetString` function, `GetIndString` procedure, and `Gestalt` function are documented in *Inside Macintosh: Operating System Utilities*.

## Packages

---

A package is a set of routines and data types that forms a part of the Toolbox or Operating System and is stored as a resource of type 'PACK'. In early models of the Macintosh computer, all packages were disk-based and brought into memory only when needed; some packages are now in ROM. The System file contains the standard Macintosh packages and the resources they use or own.

Package name	Resource ID
List Manager	0
Disk Initialization Manager	2
Standard File Package	3
Floating-Point Arithmetic Package	4
Transcendental Functions Package	5
Text Utilities	6
Text Utilities (formerly referred to as the Binary-Decimal Conversion Package)	7
Apple Event Manager	8
PPC Browser	9
Edition Manager	11
Color Picker	12
Data Access Manager	13
Help Manager	14
Picture Utilities	15



## Function Key Resources

---

Function key resources (of the 'FKEY' resource type) are Command-Shift-number key combinations that are captured and processed by the `WaitNextEvent` function. The screen utility resource (a function key resource with resource ID 3) produces a picture of the main screen, contained in a 'PICT' file, when the user presses Command-Shift-3. The 'FKEY' resource IDs 0 through 4 are reserved for future use by Apple Computer, Inc. The `WaitNextEvent` function is described in the chapter “Event Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

## Standard Icons

---









System software uses icons to represent documents, applications, folders, disks, and other elements of the Macintosh interface. Many of these standard icons are stored in the System file. You can design your own icons for your application and its documents. If you do not provide your own icons, the Finder displays a default icon. Your application can retrieve any of the icons in the System file by using the `GetResource` function. You should refer to these icons by their constant names and not by their resource IDs. For a description of the `GetResource` function, see page 1-73.

Most icons are available in at least two sizes: large (32 by 32 pixels) and small (16 by 16 pixels). They are also available in three bit depths: 8-bit, 4-bit, and black-and-white. An icon family consists of the large and small icons for an object, each with a mask, and each available in the three different color depths. See the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials* for information about how to create your own icons.

Many of the icons in the System file are also available in a small size (16 by 16 pixels), represented by the 'SICN' resource. These icons are used in Standard File Package dialog boxes. The Finder also uses icons in the System file to display in its windows the contents of disks or folders by name, date, size, or kind. The Views menu in System 7 allows the user to display large or small icons for a given window.







The icons listed in Table 1-4 represent default icons for documents (including special classes of documents such as stationery), applications, and desk accessories. The icons show the 'ic18' resource from the icons' icon family. You can include customized versions of the icons in Table 1-4 with your documents and applications. There are icon families and 'SICN' resources for all of these icons unless otherwise noted.

Table 1-4 Document and application icons

Constant name and icon	Resource ID	Description
genericDocumentIconResource 	-4000	The default document icon. The Finder displays this icon if your application does not provide its own icon for documents.
genericApplicationIconResource 	-3996	The default application icon. The Finder displays this icon for any application that does not provide its own icon.
genericDeskAccessoryIconResource 	-3991	The default desk accessory icon. In System 7 and later versions, desk accessories are represented on the desktop as applications are, each with its own icon. The Finder displays this icon for any desk accessory that does not provide its own icon.
genericEditionFileIconResource 	-3989	The default edition file icon. (See <i>Inside Macintosh: Interapplication Communication</i> for information about editions.)
genericStationeryIconResource 	-3985	The default stationery file icon. (See <i>Inside Macintosh: Macintosh Toolbox Essentials</i> for information about stationery.)
genericPreferencesIconResource 	-3971	The default preferences file icon. Preference files appear in the Preferences folder, which is located inside the System Folder. There is no 'SICN' resource for this icon.
genericQueryDocumentIconResource 	-16506	The default query document icon. (See <i>Inside Macintosh: Interapplication Communication</i> for information about query documents.) There is no 'SICN' resource for this icon.
genericExtensionIconResource 	-16415	The default extension icon. The Finder displays this icon for any extension that does not have its own icon. Extension files appear in the Extensions folder, which is located inside the System Folder. There is no 'SICN' resource for this icon.









The icons listed in Table 1-5 represent the different types of folders found on the desktop. The icons shown are the 'ic18' resource for the icons' icon families. There are icon families and 'SICN' resources for all of these icons unless otherwise noted.

**Table 1-5** Folder icons

Constant name and icon	Resource ID	Description
genericFolderIconResource 	-3999	The default folder icon.
privateFolderIconResource 	-3994	The icon for a folder to which the user does not have access. It is dimmed and has a distinctly marked border. The Finder displays an alert box when a user without privileges attempts to open this folder.
ownedFolderIconResource 	-3980	The icon for a folder that is owned by a particular user, usually on a shared volume such as a file server. There is no 'SICN' resource for this icon.
dropFolderIconResource 	-3979	The icon for a folder in which any user may store documents, applications, and so on, but from which only a specified group of users can retrieve the contents. There is no 'SICN' resource for this icon.
sharedFolderIconResource 	-3978	The icon for a folder that the owner has made available for file sharing. There is no 'SICN' resource for this icon.
mountedFolderIconResource 	-3977	The icon for a folder that a guest has mounted on a remote volume. This icon appears only for the guest. There is no 'SICN' resource for this icon.




The icons listed in Table 1-6 represent the different types of folders found in the System Folder. The icons shown are the 'ic18' resource for the icons' icon families. You should not alter the appearance of these icons. There are only icon families for these icons.

**Table 1-6** System Folder icons

Constant name and icon	Resource ID	Description
systemFolderIconResource 	-3983	The System Folder icon. This folder contains the System file and other system-related folders.
appleMenuFolderIconResource 	-3982	The Apple Menu Items folder icon. This folder contains items found in the Apple menu.
startupFolderIconResource 	-3981	The Startup Items folder icon. This folder contains documents, aliases, applications, and other objects that open when the computer starts up.
controlPanelFolderIconResource 	-3976	The Control Panels folder icon. This folder contains control panels.
printMonitorFolderIconResource 	-3975	The PrintMonitor Documents folder icon. This folder contains documents that are in the queue to be printed.
preferencesFolderIconResource 	-3974	The Preferences folder icon. This folder contains preferences files for the Finder and other software that needs to remember user preferences.
extensionsFolderIconResource 	-3973	The Extensions folder icon. This folder contains system extensions.
fontsFolderIconResource 	-3968	The Fonts folder icon. This folder contains fonts (both bitmapped and outline).



The icons listed in Table 1-7 appear on the desktop. The icons shown are the 'ic18' resource for the icons' icon families. There are icon families and 'SICN' resources for these icons unless otherwise noted.

**Table 1-7** Desktop icons

Constant name and icon	Resource ID	Description
floppyIconResource 	-3998	The default icon for any disk, 3.5-inch or otherwise, whose driver doesn't supply its own icon.
trashIconResource 	-3993	The default empty Trash icon.
fullTrashIconResource 	-3984	The default full Trash icon, with bulging midsection. There is no 'SICN' resource for this icon.




The icons listed in Table 1-8 are used only by the Standard File Package and are available only as an 'SICN' resource. The pop-up menu in the standard file dialog boxes indicates where the list of files shown in the dialog box is located (whether on the desktop, at the top level of a volume, or inside a series of folders on a volume).

**Table 1-8** Standard File Package icons

Constant name and icon	Resource ID	Description
openFolderIconResource 	-3997	The open folder icon, which appears in a pop-up menu only. The standard file dialog boxes display this icon to indicate which folder is currently open.
genericHardDiskIconResource 	-3995	The hard disk icon, which appears in a pop-up menu only. The same icon is used to represent internal and external disks. A different icon may appear on the desktop, because the manufacturer of the hard disk can design a special icon for a particular volume.

*continued*

**Table 1-8** Standard File Package icons (continued)

Constant name and icon	Resource ID	Description
desktopIconResource 	-3992	The desktop icon, which appears in a pop-up menu only. The standard file dialog boxes display this icon to indicate which files and folders are available on the desktop.
genericFileServerIconResource 	-3972	The file server volume icon. This represents any servers open on the desktop. A different icon may appear on the desktop, because the manufacturer can design a special icon for a particular server.
genericSuitcaseIconResource 	-3970	The suitcase icon. This represents any suitcase, such as font suitcases or desk accessory suitcases. There are different icons for these suitcases in larger sizes, depending on the contents.

## ROM Resources

The information in this section is useful only for designers of specialized programs that need to access ROM resources directly, bypassing any patches in the System file, or that need to override ROM resources.

### Inserting the ROM Resource Map

Many system resources are stored in ROM. System software calls the `InitResources` function during system startup, and the Resource Manager creates a special heap zone in the system heap and builds a resource map that points to the ROM resources.

The Resource Manager normally searches ROM resources only when you use the `RGetResource` function to get a handle to the resource, and even then only after it searches the System file's resource fork. To search for a resource in ROM before searching the System file's resource fork, your application must first alter the resource search order by inserting the ROM resource map in front of the System file's resource map.

When the value of the global variable `RomMapInsert` is `TRUE`, the Resource Manager inserts the ROM resource map before the System file's resource map for the next call only. When the value of `RomMapInsert` is `TRUE`, the adjacent variable `TmpResLoad` determines whether the value of the global variable `ResLoad` is considered `TRUE` or `FALSE`, overriding the actual value of `ResLoad` for the next call only. The values of the `RomMapInsert` and `TmpResLoad` variables are cleared after each call to a Resource Manager routine.

The `RGetResource` function first calls `GetResource`. If `GetResource` cannot locate the requested resource in the resource chain, `RGetResource` sets `RomMapInsert` to `TRUE`, then calls `GetResource` again.

To set the `RomMapInsert` and `TmpResLoad` variables in tandem yourself, you can use two global constants. Set the system global variable `RomMapInsert` to the global constant `mapTrue` to insert the ROM resource map with `SetResLoad(TRUE)`. Set `RomMapInsert` to the global constant `mapFalse` to insert the ROM resource map with `SetResLoad(FALSE)`.

There is no real resource fork associated with the ROM resources; the ROM resource map has a path number of 1 (an illegal path reference number). There are two ways to determine whether a handle references a ROM resource. First, you can set up `RomMapInsert` and `TmpResLoad` and call `HomeResFile`; if 1 is returned, the handle is to a ROM resource. Second, you can dereference the handle and check whether the master pointer points to ROM by comparing it to the global variable `ROMBase`.

## Overriding ROM Resources

---

You can override some of the ROM resources, such as 'CURS' resources, simply by putting the substitute resource in your application's resource fork. Other ROM resources, however, such as 'DRVr' and 'PACK' resources, cannot be overridden in this way because they are already referenced and in use when your application is launched.

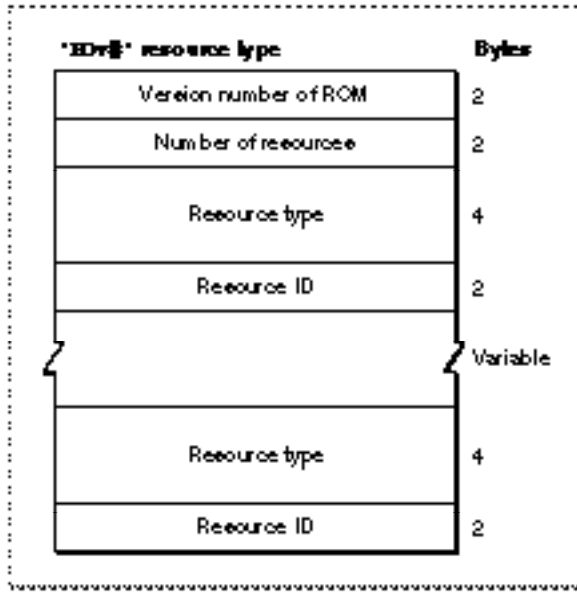
On startup, system software creates a list of ROM resources that should not be referenced. This list is based on information stored in the System file's resource fork in an 'ROv#' resource whose version word matches the version word of the ROM. You can modify the 'ROv#' resource so that it includes the ROM resources that you want to override.

### ▲ WARNING

You should not override ROM resources unless absolutely necessary. Before overriding ROM resources, you should understand the situation completely. ▲

Figure 1-19 shows the structure of an 'ROv#' resource.

**Figure 1-19** Structure of a compiled ROM override ('ROv#') resource



For information on modifying an 'ROv#' resource, write to Macintosh Developer Technical Support.



## Summary of the Resource Manager

---

### Pascal Summary

---

#### Constants

---

```

CONST
  gestaltResourceMgrAttr      = 'rsrc';    {Gestalt selector ResMgr}
  gestaltPartialRsrcs        = 0;          {check this bit in the
                                           { response parameter}

  {resource attributes}
  resSysHeap                  = 64;         {set if read into system }
                                           { heap}
  resPurgeable                = 32;         {set if purgeable}
  resLocked                   = 16;         {set if locked}
  resProtected                = 8;          {set if protected}
  resPreload                  = 4;          {set if to be preloaded}
  resChanged                  = 2;          {set if to be written to }
                                           { resource fork}

  {resource file attributes}
  mapReadOnly                 = 128;        {set to make file read-only}
  mapCompact                  = 64;         {set to compact file on }
                                           { update}
  mapChanged                  = 32;         {set to write map on update}

  {values for setting the RomMapInsert and TmpResLoad global variables}
  mapTrue                     = $FFFF;     {insert ROM map w/ }
                                           { TmpResLoad = TRUE}
  mapFalse                    = $FF00;     {insert ROM map w/ }
                                           { TmpResLoad = FALSE}

  {system icon definition IDs}
  genericDocumentIconResource = -4000;     {default document icon}
  genericFolderIconResource   = -3999;     {default folder icon}
  floppyIconResource          = -3998;     {default disk icon}
  openFolderIconResource      = -3997;     {open folder icon}
  genericApplicationIconResource = -3996;  {default application }
                                           { icon}

```

## CHAPTER 1

### Resource Manager

genericHardDiskIconResource	= -3995;	{hard disk icon}
privateFolderIconResource	= -3994;	{folder without privileges } { for this user icon}
trashIconResource	= -3993;	{default empty Trash icon}
desktopIconResource	= -3992;	{desktop icon}
genericDeskAccessoryIconResource	= -3991;	{default desk accessory icon}
genericEditionFileIconResource	= -3989;	{default edition icon}
genericStationeryIconResource	= -3985;	{default stationery icon}
systemFolderIconResource	= -3983;	{System Folder icon}
appleMenuFolderIconResource	= -3982;	{Apple Menu Items } { folder icon}
genericFileServerIconResource	= -3972;	{file server icon}
genericPreferencesIconResource	= -3971;	{default preferences } { file icon}
genericSuitcaseIconResource	= -3970;	{default suitcase icon}
genericMoverObjectIconResource	= -3969;	{System file object icon}
genericQueryDocumentIconResource	= -16506;	{default query } { document icon}
genericExtensionIconResource	= -16415;	{default extensions icon}
fullTrashIconResource	= -3984;	{default full Trash icon}
startupFolderIconResource	= -3981;	{Startup Items folder icon}
ownedFolderIconResource	= -3980;	{owned folder icon}
dropFolderIconResource	= -3979;	{drop folder icon}
sharedFolderIconResource	= -3978;	{shared folder icon}
mountedFolderIconResource	= -3977;	{mounted folder icon}
controlPanelFolderIconResource	= -3976;	{Control Panels folder icon}
printMonitorFolderIconResource	= -3975;	{PrintMonitor } { Documents folder icon}
preferencesFolderIconResource	= -3974;	{Preferences folder icon}
extensionsFolderIconResource	= -3973;	{Extensions folder icon}
fontsFolderIconResource	= -3968;	{Fonts folder icon}

## Data Type

---

```
TYPE ResType = PACKED ARRAY[1..4] OF Char;
```

## Routines

---

### Initializing the Resource Manager

```
FUNCTION InitResources:      Integer;
PROCEDURE RsrcZoneInit;
```

### Checking for Errors

```
FUNCTION ResError:          Integer;
```

### Creating an Empty Resource Fork

```
PROCEDURE FSpCreateResFile (spec: FSSpec; creator: OSType;
                           fileType: OSType; scriptTag: ScriptCode);
PROCEDURE HCreateResFile  (vRefNum: Integer; dirID: LongInt;
                           fileName: Str255);
PROCEDURE CreateResFile   (fileName: Str255);
```

### Opening Resource Forks

```
FUNCTION FSpOpenResFile    (spec: FSSpec; permission: SignedByte): Integer;
FUNCTION HOpenResFile     (vRefNum: Integer; dirID: LongInt;
                           fileName: Str255;
                           permission: SignedByte): Integer;
FUNCTION OpenRFPPerm      (fileName: Str255; vRefNum: Integer;
                           permission: SignedByte): Integer;
FUNCTION OpenResFile      (fileName: Str255): Integer;
```

### Getting and Setting the Current Resource File

```
FUNCTION CurResFile:      Integer;
PROCEDURE UseResFile     (refNum: Integer);
FUNCTION HomeResFile     (theResource: Handle): Integer;
```

### Reading Resources Into Memory

```
FUNCTION GetResource      (theType: ResType; theID: Integer): Handle;  
FUNCTION Get1Resource    (theType: ResType; theID: Integer): Handle;  
FUNCTION GetNamedResource (theType: ResType; name: Str255): Handle;  
FUNCTION Get1NamedResource (theType: ResType; name: Str255): Handle;  
FUNCTION RGetResource    (theType: ResType; theID: Integer): Handle;  
PROCEDURE SetResLoad     (load: Boolean);  
PROCEDURE LoadResource   (theResource: Handle);
```

### Getting and Setting Resource Information

```
PROCEDURE GetResInfo      (theResource: Handle; VAR theID: Integer;  
                          VAR theType: ResType; VAR name: Str255);  
PROCEDURE SetResInfo      (theResource: Handle; theID: Integer;  
                          name: Str255);  
FUNCTION GetResAttrs      (theResource: Handle): Integer;  
PROCEDURE SetResAttrs     (theResource: Handle; attrs: Integer);
```

### Modifying Resources

```
PROCEDURE ChangedResource (theResource: Handle);  
PROCEDURE AddResource     (theData: Handle; theType: ResType;  
                          theID: Integer; name: Str255);
```

### Writing to Resource Forks

```
PROCEDURE UpdateResFile   (refNum: Integer);  
PROCEDURE WriteResource   (theResource: Handle);  
PROCEDURE SetResPurge     (install: Boolean);
```

### Getting a Unique Resource ID

```
FUNCTION UniqueID         (theType: ResType): Integer;  
FUNCTION Unique1ID       (theType: ResType): Integer;
```

### Counting and Listing Resource Types

```
FUNCTION CountResources   (theType: ResType): Integer;  
FUNCTION Count1Resources  (theType: ResType): Integer;  
FUNCTION GetIndResource   (theType: ResType; index: Integer): Handle;  
FUNCTION Get1IndResource  (theType: ResType; index: Integer): Handle;
```

```

FUNCTION CountTypes:      Integer;
FUNCTION Count1Types:    Integer;
PROCEDURE GetIndType     (VAR theType: ResType; index: Integer);
PROCEDURE Get1IndType    (VAR theType: ResType; index: Integer);

```

### Getting Resource Sizes

{these routines also available as SizeResource and MaxSizeRsrc, respectively}

```

FUNCTION GetResourceSizeOnDisk
                                (theResource: Handle): LongInt;
FUNCTION GetMaxResourceSize
                                (theResource: Handle): LongInt;

```

### Disposing of Resources

```

PROCEDURE ReleaseResource (theResource: Handle);
PROCEDURE DetachResource  (theResource: Handle);
{The RemoveResource procedure is also available as RmveResource}
PROCEDURE RemoveResource  (theResource: Handle);

```

### Closing Resource Forks

```

PROCEDURE CloseResFile      (refNum: Integer);

```

### Reading and Writing Partial Resources

```

PROCEDURE ReadPartialResource
                                (theResource: Handle;
                                 offset: LongInt; buffer: UNIV Ptr;
                                 count: LongInt);
PROCEDURE WritePartialResource
                                (theResource: Handle;
                                 offset: LongInt; buffer: UNIV Ptr;
                                 count: LongInt);
PROCEDURE SetResourceSize (theResource: Handle; newSize: LongInt);

```

### Getting and Setting Resource Fork Attributes

```

FUNCTION GetResFileAttrs (refNum: Integer): Integer;
PROCEDURE SetResFileAttrs (refNum: Integer; attrs: Integer);

```

### Accessing Resource Entries in a Resource Map

```

FUNCTION RsrcMapEntry      (theResource: Handle): LongInt;

```

## C Summary

---

### Constants

---

```

enum {
#define gestaltResourceMgrAttr      'rsrc'  /*Gestalt selector ResMgr*/
#define gestaltPartialRsrcs        = 0      /*check this bit in the */
                                        /* response parameter*/
};
enum {
    /*resource attributes*/
    resSysHeap                      = 64,    /*set if read into system heap*/
    resPurgeable                    = 32,    /*set if purgeable*/
    resLocked                        = 16,    /*set if locked*/
    resProtected                     = 8,     /*set if protected*/
    resPreload                       = 4,    /*set if to be preloaded*/
    resChanged                       = 2,    /*set if to be written */
                                        /* to resource fork*/

    /*resource fork attributes*/
    mapReadOnly                      = 128,   /*set to make file */
                                        /* read-only*/
    mapCompact                       = 64,    /*set to compact file */
                                        /* on update*/
    mapChanged                       = 32,    /*set to write map */
                                        /* on update*/

    /*values for setting the RomMapInsert and TmpResLoad global variables*/
    mapTrue                          = 0xFFFF, /*insert ROM map w/ */
                                        /* TmpResLoad = TRUE*/
    mapFalse                         = 0xFF00 /*insert ROM map w/ */
                                        /* TmpResLoad = FALSE*/
};
enum {
    /*system icon definition IDs*/
    genericDocumentIconResource      = -4000, /*default document icon*/
    genericStationeryIconResource    = -3985, /*default stationery icon*/
    genericEditionFileIconResource   = -3989, /*default edition icon*/
    genericApplicationIconResource   = -3996, /*default application icon*/
    genericDeskAccessoryIconResource = -3991, /*default desk accessory */
                                        /* icon*/
    genericFolderIconResource        = -3999, /*default folder icon*/
    privateFolderIconResource        = -3994, /*folder without privileges*/
                                        /* for this user icon*/
};

```

```

floppyIconResource          = -3998,    /*default disk icon*/
trashIconResource          = -3993,    /*default empty Trash icon*/
desktopIconResource        = -3992,    /*desktop icon*/
openFolderIconResource     = -3997,    /*open folder icon*/
genericHardDiskIconResource = -3995,    /*hard disk icon*/
genericFileServerIconResource = -3972,    /*file server icon*/
genericSuitcaseIconResource = -3970,    /*default suitcase icon*/
genericMoverObjectIconResource = -3969,    /*System file object icon*/
genericPreferencesIconResource = -3971,    /*default preferences */
                           /* file icon*/

genericQueryDocumentIconResource = -16506,    /*default query doc icon*/
genericExtensionIconResource = -16415,    /*default extension icon*/
systemFolderIconResource   = -3983,    /*System Folder icon*/
appleMenuFolderIconResource = -3982,    /*Apple Menu Items */
                           /* folder icon*/

};

enum {
    startupFolderIconResource = -3981,    /*Startup Items folder icon*/
    ownedFolderIconResource   = -3980,    /*owned folder icon*/
    dropFolderIconResource    = -3979,    /*drop folder icon*/
    sharedFolderIconResource  = -3978,    /*shared folder icon*/
    mountedFolderIconResource = -3977,    /*mounted folder icon*/
    controlPanelFolderIconResource = -3976,    /*Control Panels folder */
                           /* icon*/

    printMonitorFolderIconResource = -3975,    /*PrintMonitor */
                           /* Documents folder icon*/

    preferencesFolderIconResource = -3974,    /*Preferences folder icon*/
    extensionsFolderIconResource  = -3973,    /*Extensions folder icon*/
    fontsFolderIconResource      = -3968,    /*Fonts folder icon*/
    fullTrashIconResource        = -3984    /*default full Trash icon*/
};

```

## Data Type

---

```
typedef unsigned long ResType;
```

Routines

---

**Initializing the Resource Manager**

```
pascal short InitResources (void);
pascal void RsrcZoneInit (void);
```

**Checking for Errors**

```
pascal short ResError (void);
```

**Creating an Empty Resource Fork**

```
pascal void FSpCreateResFile
                                (const FSSpec *spec, OSType creator,
                                 OSType fileType, ScriptCode scriptTag);
pascal void HCreateResFile (short vRefNum, long dirID,
                            ConstStr255Param fileName);
pascal void CreateResFile (ConstStr255Param fileName);
```

**Opening Resource Forks**

```
pascal short FSpOpenResFile
                                (const FSSpec *spec, SignedByte permission);
pascal short HOpenResFile (short vRefNum, long dirID,
                            ConstStr255Param fileName,
                            char permission);
pascal short OpenRFPPerm (ConstStr255Param fileName, short vRefNum,
                           char permission);
pascal short OpenResFile (ConstStr255Param fileName);
```

**Getting and Setting the Current Resource File**

```
pascal short CurResFile (void);
pascal void UseResFile (short refNum);
pascal short HomeResFile (Handle theResource);
```

**Reading Resources Into Memory**

```
pascal Handle GetResource (ResType theType, short theID);
pascal Handle Get1Resource (ResType theType, short theID);
pascal Handle GetNamedResource
                                (ResType theType, ConstStr255Param name);
pascal Handle Get1NamedResource
                                (ResType theType, ConstStr255Param name);
```



```

pascal Handle RGetResource (ResType theType, short theID);
pascal void SetResLoad (Boolean load);
pascal void LoadResource (Handle theResource);

```

### Getting and Setting Resource Information

```

pascal void GetResInfo (Handle theResource, short *theID,
                       ResType *theType, Str255 name);
pascal void SetResInfo (Handle theResource, short theID,
                       ConstStr255Param name);
pascal short GetResAttrs (Handle theResource);
pascal void SetResAttrs (Handle theResource, short attrs);

```

### Modifying Resources

```

pascal void ChangedResource (Handle theResource);
pascal void AddResource (Handle theData, ResType theType,
                        short theID, ConstStr255Param name);

```

### Writing to Resource Forks

```

pascal void UpdateResFile (short refNum);
pascal void WriteResource (Handle theResource);
pascal void SetResPurge (Boolean install);

```

### Getting a Unique Resource ID

```

pascal short UniqueID (ResType theType);
pascal short Unique1ID (ResType theType);

```

### Counting and Listing Resource Types

```

pascal short CountResources (ResType theType);
pascal short Count1Resources (ResType theType);
pascal Handle GetIndResource (ResType theType, short index);
pascal Handle Get1IndResource (ResType theType, short index);
pascal short CountTypes (void);
pascal short Count1Types (void);

```

## CHAPTER 1

### Resource Manager

```
pascal void GetIndType      (ResType *theType, short index);
pascal void Get1IndType    (ResType *theType, short index);
```

#### Getting Resource Sizes

```
/*the GetResourceSizeOnDisk routine is also available as SizeResource*/
pascal long GetResourceSizeOnDisk
                (Handle theResource);
/*the GetMaxResourceSize routine is also available as MaxSizeRsrc*/
pascal long GetMaxResourceSize
                (Handle theResource);
```

#### Disposing of Resources

```
pascal void ReleaseResource
                (Handle theResource);
pascal void DetachResource  (Handle theResource);
/*the RemoveResource routine is also available as RvmeResource*/
pascal void RemoveResource  (Handle theResource);
```

#### Closing Resource Forks

```
pascal void CloseResFile   (short refNum);
```

#### Reading and Writing Partial Resources

```
pascal void ReadPartialResource
                (Handle theResource, long offset,
                void *buffer, long count);
pascal void WritePartialResource
                (Handle theResource, long offset,
                const void *buffer, long count);
pascal void SetResourceSize
                (Handle theResource, long newSize);
```

#### Getting and Setting Resource Fork Attributes

```
pascal short GetResFileAttrs
                (short refNum);
pascal void SetResFileAttrs
                (short refNum, short attrs);
```

**Accessing Resource Entries in a Resource Map**

```
pascal long RsrcMapEntry    (Handle theResource);
```

**Assembly-Language Summary**

---

**Trap Macros**

---

**Trap Macros Requiring Routine Selectors**`_ResourceDispatch`

<b>Selector</b>	<b>Routine</b>
\$7001	ReadPartialResource
\$7002	WritePartialResource
\$7003	SetResourceSize

`_HighLevelFSDispatch`

<b>Selector</b>	<b>Routine</b>
\$0000	FSpOpenResFile
\$000E	FSpCreateResFile

**Global Variables**

---

TopMapHndl	long	Handle to resource map of most recently opened resource fork
SysMapHndl	long	Handle to System file's resource fork
SysMap	word	File reference number of System file's resource fork
CurMap	word	File reference number of current resource file
ResLoad	word	Current SetResLoad state
ResErr	word	Current value of ResError
ResErrProc	long	Address of resource error procedure
SysResName	length byte followed by up to 19 characters	Name of System file's resource fork
RomMapInsert	byte	Flag for whether to insert ROM resource map
TmpResLoad	byte	Temporary SetResLoad state for calls using RomMapInsert

## Result Codes

---

noErr	0	No error
dirFulErr	-33	Directory full
dskFulErr	-34	Disk full
nsvErr	-35	No such volume
ioErr	-36	I/O error
bdNamErr	-37	Bad filename or volume name (perhaps zero length)
eofErr	-39	End of file
tmfoErr	-42	Too many files open
fnfErr	-43	File not found
wPrErr	-44	Disk is write-protected
fLckdErr	-45	File is locked
vLckdErr	-46	Volume is locked
dupFNerr	-48	Duplicate filename (rename)
opWrErr	-49	File already open with write permission
permErr	-54	Permissions error (on file open)
extFSerr	-58	Volume belongs to an external file system
memFullErr	-108	Not enough room in heap zone
dirNFerr	-120	Directory not found
resourceInMemory	-188	Resource already in memory
writingPastEnd	-189	Writing past end of file
inputOutOfBounds	-190	Offset or count out of bounds
resNotFound	-192	Resource not found
resFNotFound	-193	Resource file not found
addResFailed	-194	AddResource procedure failed
rmvResFailed	-196	RemoveResource procedure failed
resAttrErr	-198	Attribute inconsistent with operation
mapReadErr	-199	Map inconsistent with operation