# Resource Manager Reference

**Carbon > File Management**

**2007-10-31**

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010



Simultaneously published in the United States and Canada.

# Contents


## Resource Manager Reference 7

Overview 7
Functions by Task 7
- Checking for Errors 7
- Closing Resource Forks 7
- Counting and Listing Resource Types 7
- Creating Resource Files and Forks 8
- Disposing of Resources 8
- Getting a Unique Resource ID 9
- Getting and Setting Resource Fork Attributes 9
- Getting and Setting Resource Information 9
- Getting and Setting the Current Resource File 9
- Getting Resource Sizes 9
- Managing the Resource Chain 10
- Modifying Resources 10
- Opening Resource Forks 10
- Reading and Writing Partial Resources 11
- Reading Resources Into Memory 11
- Writing to Resource Forks 11
- Not Recommended 11

Functions 12
- AddResource 12
- ChangedResource 13
- CloseResFile 14
- Count1Resources 15
- Count1Types 15
- CountResources 16
- CountTypes 16
- CurResFile 16
- DetachResource 17
- DetachResourceFile 18
- DisposeResErrUPP 18
- FSCreateResFile 18
- FSCreateResourceFile 19
- FSCreateResourceFork 20
- FSOpenOrphanResFile 21
- FSOpenResFile 21
- FSOpenResourceFile 22
- FSResourceFileAlreadyOpen 22
- Get1IndResource 23
- Get1IndType 24

Get1NamedResource 24
Get1Resource 25
GetIndResource 26
GetIndType 27
GetMaxResourceSize 28
GetNamedResource 28
GetNextFOND 29
GetNextResourceFile 29
GetResAttrs 30
GetResFileAttrs 30
GetResInfo 31
GetResource 31
GetResourceSizeOnDisk 32
GetTopResourceFile 33
HomeResFile 33
InsertResourceFile 34
InvokeResErrUPP 34
LoadResource 35
NewResErrUPP 35
ReadPartialResource 36
ReleaseResource 37
RemoveResource 37
ResError 38
SetResAttrs 39
SetResFileAttrs 39
SetResInfo 40
SetResLoad 41
SetResourceSize 42
SetResPurge 43
Unique1ID 43
UniqueID 44
UpdateResFile 45
UseResFile 46
WritePartialResource 47
WriteResource 48
Callbacks 49
ResErrProcPtr 49
ResourceEndianFilterPtr 49
Data Types 49
ResAttributes 49
ResErrUPP 50
ResFileAttributes 50
ResFileRefNum 50
ResID 50
ResType 51
Constants 51

Reference Number Constants 51
Resource Attribute Bits 52
Resource Attribute Masks 53
Resource Chain Location 54
Resource Fork Attribute Bits 55
Resource Fork Attribute Masks 56
Result Codes 56

Appendix A Deprecated Resource Manager Functions 59

Deprecated in Mac OS X v10.5 59
FSpCreateResFile 59
FSpOpenOrphanResFile 59
FSpOpenResFile 60
FSpResourceFileAlreadyOpen 62
HCreateResFile 62
HOpenResFile 63
OpenRFPerm 64

Document Revision History 67

Index 69

# Resource Manager Reference

| | |
|---|---|
| **Framework:** | CoreServices/CoreServices.h |
| **Declared in** | IOMacOSTypes.h<br>Resources.h |

## Overview

The Resource Manager allows applications to create, delete, open, read, modify, and write resources, get information about them, and alter the Resource Manager's search path. A resource is data of any kind stored in a defined format in a resource file. The Resource Manager keeps track of resources in memory and provides functions for the proper management of the resource chain. In Mac OS X, you should store resources in the data fork of a resource file.

Carbon applications have used Resource Manager resources to store the descriptions for user interface elements such as menus, windows, dialogs, controls, and icons. In addition, applications have used resources to store variable settings, such as the location of a document window at the time the user closes the window. When the user opens the document again, the application reads the information in the appropriate resource and restores the window to its previous location.

## Functions by Task

### Checking for Errors

`ResError` (page 38)
Determines what error occurred, if any, after calling a Resource Manager function.

### Closing Resource Forks

`CloseResFile` (page 14)
Closes a resource fork before your application terminates.

### Counting and Listing Resource Types

`Count1Resources` (page 15)
Returns the total number of resources of a given type in the current resource file.

`Count1Types` (page 15)
Returns the number of resource types in the current resource file.

`CountResources` (page 16)
Returns the total number of available resources of a given type.

`CountTypes` (page 16)
Returns the number of resource types in all resource forks open to your application.

`Get1IndResource` (page 23)
Returns a handle to a resource of a given type in the current resource file.

`Get1IndType` (page 24)
Gets a resource types available in the current resource file.

`GetIndResource` (page 26)
Returns a handle to a resource of a given type in resource forks open to your application.

`GetIndType` (page 27)
Gets a resource type available in resource forks open to your application.

## Creating Resource Files and Forks

`FSCreateResourceFile` (page 19)
Creates a file with a named fork for storing resource data.

`FSCreateResourceFork` (page 20)
Creates a named fork for storing resource data.

`FSCreateResFile` (page 18)
Creates a file with an empty resource fork.

`FSpCreateResFile` (page 59) Deprecated in Mac OS X v10.5
Creates an empty resource fork in a new or existing file. (Deprecated. Use `FSCreateResourceFile` (page 19) instead.)

`HCreateResFile` (page 62) Deprecated in Mac OS X v10.5
Creates an empty resource fork, when the `FSpCreateResFile` function is not available. (Deprecated. Use `FSCreateResourceFile` (page 19) instead.)

## Disposing of Resources

`DetachResource` (page 17)
Sets the value of a resource's handle in the resource map in memory to `NULL` while keeping the resource data in memory.

`ReleaseResource` (page 37)
Releases the memory a resource occupies when you have finished using it.

`RemoveResource` (page 37)
Removes a resource's entry from the current resource file's resource map in memory.

## Getting a Unique Resource ID

`UniqueID` (page 43)
Gets a resource ID that's unique with respect to resources in the current resource file.

`UniqueID` (page 44)
Gets a unique resource ID for a resource.

## Getting and Setting Resource Fork Attributes

`GetResFileAttrs` (page 30)
Gets the attributes of a resource fork.

`SetResFileAttrs` (page 39)
Sets a resource fork's attributes.

## Getting and Setting Resource Information

`GetResAttrs` (page 30)
Gets a resource's attributes.

`GetResInfo` (page 31)
Gets a resource's resource ID, resource type, and resource name.

`SetResAttrs` (page 39)
Sets a resource's attributes in the resource map in memory.

`SetResInfo` (page 40)
Sets the name and resource ID of a resource.

## Getting and Setting the Current Resource File

`CurResFile` (page 16)
Gets the file reference number of the current resource file.

`HomeResFile` (page 33)
Gets the file reference number associated with a particular resource.

`UseResFile` (page 46)
Sets the current resource file.

## Getting Resource Sizes

`GetMaxResourceSize` (page 28)
Returns the approximate size of a resource.

`GetResourceSizeOnDisk` (page 32)
Returns the exact size of a resource.

## Managing the Resource Chain

`InsertResourceFile` (page 34)
Inserts a resource file into the current resource chain at the specified location.

`DetachResourceFile` (page 18)
Removes a resource file from the resource chain.

`GetNextResourceFile` (page 29)
Retrieves the next resource file in the resource chain.

`GetTopResourceFile` (page 33)
Retrieves the topmost resource file in the current resource chain.

## Modifying Resources

`AddResource` (page 12)
Adds a resource to the current resource file's resource map in memory.

`ChangedResource` (page 13)
Sets a flag in the resource's resource map entry in memory to show that you've made changes to a resource's data or to an entry in a resource map.

## Opening Resource Forks

`FSOpenResourceFile` (page 22)
Opens a named fork in an existing resource file.

`FSOpenOrphanResFile` (page 21)
Opens a resource file that is persistent across all contexts.

`FSOpenResFile` (page 21)
Opens the resource fork in a file specified with an `FSRef` structure.

`FSResourceFileAlreadyOpen` (page 22)
Checks whether a resource file is open.

`FSpOpenOrphanResFile` (page 59) Deprecated in Mac OS X v10.5
Opens a resource file that is persistent across all contexts. (Deprecated. Use `FSOpenOrphanResFile` (page 21) instead.)

`FSpOpenResFile` (page 60) Deprecated in Mac OS X v10.5
Opens the resource fork in a file specified with an `FSSpec` structure. (Deprecated. Use `FSOpenResourceFile` (page 22) instead.)

`FSpResourceFileAlreadyOpen` (page 62) Deprecated in Mac OS X v10.5
Checks whether a resource file is open. (Deprecated. Use `FSResourceFileAlreadyOpen` (page 22) instead.)

`HOpenResFile` (page 63) Deprecated in Mac OS X v10.5
Opens a file's resource fork, when the `FSpOpenResFile` function is not available. (Deprecated. Use `FSOpenResourceFile` (page 22) instead.)

`OpenRFPerm` (page 64) Deprecated in Mac OS X v10.5
Opens a file's resource fork, when the `FSpOpenResFile` and `HOpenResFile` functions are not available. (Deprecated. Use `FSOpenResourceFile` (page 22) instead.)

## Reading and Writing Partial Resources

`ReadPartialResource` (page 36)
Reads part of a resource into memory and work with a small subsection of a large resource.

`SetResourceSize` (page 42)
Sets the size of a resource on disk.

`WritePartialResource` (page 47)
Writes part of a resource to disk when working with a small subsection of a large resource.

## Reading Resources Into Memory

`Get1NamedResource` (page 24)
Gets a named resource in the current resource file.

`Get1Resource` (page 25)
Gets resource data for a resource in the current resource file.

`GetNamedResource` (page 28)
Gets a named resource.

`GetResource` (page 31)
Gets resource data for a resource specified by resource type and resource ID.

`LoadResource` (page 35)
Gets resource data after you've called the `SetResLoad` function with the `load` parameter set to `FALSE` or when the resource is purgeable.

`SetResLoad` (page 41)
Enables and disables automatic loading of resource data into memory for functions that return handles to resources.

## Writing to Resource Forks

`SetResPurge` (page 43)
Tells the Memory Manager to pass the handle of a resource to the Resource Manager before purging the data specified by that handle.

`UpdateResFile` (page 45)
Updates the resource map and resource data for a resource fork without closing it.

`WriteResource` (page 48)
Writes resource data in memory immediately to a file's resource fork.

## Not Recommended

`GetNextFOND` (page 29)
Gets the next FOND handle.

`InvokeResErrUPP` (page 34)
Calls your callback function.

`NewResErrUPP` (page 35)
Creates a new universal procedure pointer (UPP) to your callback function.

`DisposeResErrUPP` (page 18)
Disposes of the universal procedure pointer (UPP) to your callback function.

# Functions

## AddResource

Adds a resource to the current resource file's resource map in memory.

```
void AddResource (
   Handle theData,
   ResType theType,
   ResID theID,
   ConstStr255Param name
);
```

### Parameters

*theData*
A handle to data in memory to be added as a resource to the current resource file (not a handle to an existing resource). If the value of this parameter is an empty handle (that is, a handle whose master pointer is set to `NULL`), the Resource Manager writes zero-length resource data to disk when it updates the resource. If its value is either `NULL` or a handle to an existing resource, the function does nothing, and the `ResError` (page 38) function returns the result code `addResFailed`. If the resource map becomes too large to fit in memory, the function 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).

*theType*
The resource type of the resource to be added.

*theID*
The resource ID of the resource to be added.

*name*
The name of the resource to be added.

### Discussion

This function 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. If the `resChanged` attribute of a resource has been set and your application calls the `UpdateResFile` (page 45) function 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 the `WriteResource` (page 48) function, 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.

The function 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` (page 44) or `Unique1ID` (page 43) function to get a unique resource ID before adding a resource with this function.

When your application calls this function, 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 function 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` (page 17) function, 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.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## ChangedResource

Sets a flag in the resource's resource map entry in memory to show that you've made changes to a resource's data or to an entry in a resource map.

```
void ChangedResource (
   Handle theResource
);
```

**Parameters**

*`theResource`*

A handle to the resource whose data you've changed. The function 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 the `UpdateResFile` (page 45) function 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 the `WriteResource` (page 48) function, the Resource Manager writes only the resource data for that resource to disk.

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, the function does nothing. To find out whether any of these errors occurred, call the `ResError` (page 38) function.

**Discussion**
If you change information in the resource map with a call to the `SetResInfo` (page 40) or `SetResAttrs` (page 39) function 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.

When your application calls this function, the Resource Manager attempts to reserve enough disk space to contain the changed resource. The function 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 function 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 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 function 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.

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

If you need to make changes to a purgeable resource using functions 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 functions 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 function 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 the SetResPurge (page 43) function 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 functions that may cause the resource to be purged, you don't need to take these precautions

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## CloseResFile

Closes a resource fork before your application terminates.

```
void CloseResFile (
   ResFileRefNum refNum
);
```

**Parameters**

*refNum*

The file reference number for the resource fork to close. If this parameter does not contain a file reference number for a file whose resource fork is open, the function does nothing, and the ResError (page 38) function returns the result code resFNotFound. If the value of this parameter is 0, it represents the System file and is ignored. You cannot close the System file's resource fork.

**Discussion**
This function performs four tasks. First, it updates the file by calling the UpdateResFile (page 45) function. Second, it releases the memory occupied by each resource in the resource fork by calling the DisposeHandle function. Third, it releases the memory occupied by the resource map. The fourth task is to close the 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 this function only if you want to close a resource fork before your application terminates.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
Simple DrawSprocket

**Declared In**
`Resources.h`

## Count1Resources

Returns the total number of resources of a given type in the current resource file.

```
ResourceCount Count1Resources (
   ResType theType
);
```

**Parameters**

*theType*

The resource type of the resources to count.

**Return Value**
The total number of resources of the given type in the current resource file.

**Discussion**
To check for errors, call the `ResError` (page 38) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## Count1Types

Returns the number of resource types in the current resource file.

```
ResourceCount Count1Types (
   void
);
```

**Return Value**
The total number of unique resource types in the current resource file.

**Discussion**
To check for errors, call the `ResError` (page 38) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## CountResources

Returns the total number of available resources of a given type.

```
ResourceCount CountResources (
   ResType theType
);
```

**Parameters**

*theType*
A resource type.

**Return Value**
The total number of resources of the given type in all resource forks open to your application.

**Discussion**
To check for errors, call the ResError (page 38) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## CountTypes

Returns the number of resource types in all resource forks open to your application.

```
ResourceCount CountTypes (
   void
);
```

**Return Value**
The total number of unique resource types in all resource forks open to your application.

**Discussion**
To check for errors, call the ResError (page 38) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## CurResFile

Gets the file reference number of the current resource file.

```
ResFileRefNum CurResFile (
   void
);
```

**Return Value**
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, the function returns the actual file reference number. You can use this number or 0 with functions that take a file reference number for the System file. All Resource Manager functions recognize both 0 and the actual file reference number as referring to the System file.

**Discussion**
Most of the Resource Manager functions 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.

To check for errors, call the `ResError` (page 38) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
Simple DrawSprocket

**Declared In**
`Resources.h`

## DetachResource

Sets the value of a resource's handle in the resource map in memory to `NULL` while keeping the resource data in memory.

```
void DetachResource (
   Handle theResource
);
```

**Parameters**

*`theResource`*

A handle to the resource which you wish to detach. If this parameter doesn't contain a handle to a resource or if the resource's `resChanged` attribute is set, the function does nothing. To determine whether either of these errors occurred, call the `ResError` (page 38) function.

**Discussion**
After this call, the Resource Manager no longer recognizes the handle as a handle to a resource. However, this function 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 function to get the released resource, the Resource Manager assigns a new handle. You can use `DetachResource` if you want to access the resource data directly without using Resource Manager functions. You can also use the `DetachResource` function 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 the `AddResource` (page 12) function using a different resource ID.

**Special Considerations**
Do not use this function to detach a System resource that might be shared by several applications.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
QTMetaData

**Declared In**
`Resources.h`

## DetachResourceFile

Removes a resource file from the resource chain.

```
OSErr DetachResourceFile (
   ResFileRefNum refNum
);
```

**Return Value**
A result code. See "Resource Manager Result Codes" (page 56).

**Discussion**
If the file is not currently in the resource chain, this returns `resNotFound`. Otherwise, the resource file is removed from the resource chain.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## DisposeResErrUPP

Disposes of the universal procedure pointer (UPP) to your callback function.

```
void DisposeResErrUPP (
    ResErrUPP userUPP
);
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## FSCreateResFile

Creates a file with an empty resource fork.

```
void FSCreateResFile (
    const FSRef *parentRef,
    UniCharCount nameLength,
    const UniChar *name,
    FSCatalogInfoBitmap whichInfo,
    const FSCatalogInfo *catalogInfo,
    FSRef *newRef,
    FSSpec *newSpec
);
```

**Discussion**
This function is not recommended. You should use a file's data fork instead of its resource fork to store resource data.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## FSCreateResourceFile

Creates a file with a named fork for storing resource data.

```
OSErr FSCreateResourceFile (
   const FSRef *parentRef,
   UniCharCount nameLength,
   const UniChar *name,
   FSCatalogInfoBitmap whichInfo,
   const FSCatalogInfo *catalogInfo,
   UniCharCount forkNameLength,
   const UniChar *forkName,
   FSRef *newRef,
   FSSpec *newSpec
);
```

**Parameters**

*parentRef*
: A pointer to the directory in which the resource file is to be created.

*nameLength*
: The number of Unicode characters in the file's name.

*name*
: A pointer to the Unicode name of the new resource file.

*whichInfo*
: The catalog information fields to set. See the File Manager documentation for a description of the `FSCatalogInfoBitmap` data type.

*catalogInfo*
: A pointer to the values for the catalog information fields. This pointer may be set to `NULL`. See the File Manager documentation for a description of the `FSCatalogInfo` data type.

*forkNameLength*
: The number of Unicode characters in the fork's name.

*forkName*

A pointer to the Unicode name of the fork to initialize. If you pass `NULL` in this parameter, the data fork is used.

*newRef*

A pointer to a variable allocated by the caller, or `NULL`. On return, the new resource file.

*newSpec*

A pointer to a variable allocated by the caller, or `NULL`. On return, the new resource file.

**Return Value**
A result code. See "Resource Manager Result Codes" (page 56).

**Discussion**
This function creates a new file and initializes the specified fork for storing resource data. If you don't specify the fork name, the data fork is used. This function makes it possible to store resources in the data fork of a file.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## FSCreateResourceFork

Creates a named fork for storing resource data.

```
OSErr FSCreateResourceFork (
   const FSRef *ref,
   UniCharCount forkNameLength,
   const UniChar *forkName,
   UInt32 flags
);
```

**Parameters**

*ref*

A pointer to the file to which to add the fork.

*forkNameLength*

The number of Unicode characters in the fork's name.

*forkName*

A pointer to the Unicode name of the fork to initialize. If you pass `NULL` in this parameter, the data fork is used.

*flags*

A value of type `UInt32`. You should pass `0`.

**Return Value**
A result code. See "Resource Manager Result Codes" (page 56).

**Discussion**
This function creates the specified fork in an existing file and initializes the fork for storing resources. If the named fork already exists, this function does nothing and returns `errFSForkExists`. If you don't specify the fork name, the data fork is used. This function makes it possible to store resources in the data fork of a file.

**Availability**
Available in Mac OS X v10.2 and later.

**Declared In**
`Resources.h`

## FSOpenOrphanResFile

Opens a resource file that is persistent across all contexts.

```
OSErr FSOpenOrphanResFile (
    const FSRef *ref,
    SignedByte permission,
    ResFileRefNum *refNum
);
```

**Parameters**

*ref*

A pointer to the resource file to open.

*permission*

A constant indicating the type of access with which to open the resource fork. For a description of the types of access you can request, see File Access Permission Constants in *File Manager Reference*.

*refNum*

A pointer to a variable allocated by the caller. On return, the reference number for accessing the open fork.

**Return Value**
A result code. See "Resource Manager Result Codes" (page 56).

**Discussion**
This function loads a map and all preloaded resources into the system context and detaches the specified file from the context in which it was opened. If the file is already in the resource chain and a new instance is not opened, this function returns `paramErr`. Use this function with care, as it may fail if the map is very large or many resources are preloaded.

**Availability**
Available in Mac OS X v10.5 and later.

**Declared In**
`Resources.h`

## FSOpenResFile

Opens the resource fork in a file specified with an `FSRef` structure.

```
ResFileRefNum FSOpenResFile (
    const FSRef *ref,
    SInt8 permission
);
```

**Discussion**
This function is not recommended. You should use a file's data fork instead of its resource fork to store resource data.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## FSOpenResourceFile

Opens a named fork in an existing resource file.

```
OSErr FSOpenResourceFile (
   const FSRef *ref,
   UniCharCount forkNameLength,
   const UniChar *forkName,
   SInt8 permissions,
   ResFileRefNum *refNum
);
```

**Parameters**

*ref*

A pointer to the file containing the fork to open.

*forkNameLength*

The number of Unicode characters in the fork's name.

*forkName*

A pointer to the Unicode name of the fork to open. If you pass `NULL` in this parameter, the data fork is used.

*permissions*

A constant indicating the type of access with which to open the fork. For a description of the types of access you can request, see File Access Permission Constants in *File Manager Reference*.

*refNum*

A pointer to a variable allocated by the caller. On return, the reference number for accessing the open fork.

**Return Value**
A result code. See "Resource Manager Result Codes" (page 56).

**Discussion**
This function allows any named fork of a file to be used for storing resources. Passing in a null fork name will result in the data fork being used. You should use a file's data fork to store resource data.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## FSResourceFileAlreadyOpen

Checks whether a resource file is open.

```
Boolean FSResourceFileAlreadyOpen (
   const FSRef *resourceFileRef,
   Boolean *inChain,
   ResFileRefNum *refNum
);
```

**Parameters**

*resourceFileRef*

The resource file to check.

*inChain*

A pointer to a variable allocated by the caller. On return, `true` if the resource file is in the resource chain, `false` otherwise.

*refNum*

A pointer to a variable allocated by the caller. On return, the reference number of the file if it is open.

**Return Value**

This function returns `true` if the resource file is already open and known by the Resource Manager—for example, it is either in the current resource chain or it is a detached resource file.

**Availability**

Available in Mac OS X v10.0 and later.

**Declared In**

`Resources.h`

## Get1IndResource

Returns a handle to a resource of a given type in the current resource file.

```
Handle Get1IndResource (
   ResType theType,
   ResourceIndex index
);
```

**Parameters**

*theType*

A resource type.

*index*

An integer ranging from 1 to the number of resources of a given type returned by the `Count1Resources` (page 15) function, which is the number of resources of that type in the current resource file.

**Return Value**

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. If you provide an index that is either 0 or negative, the function returns `NULL`, and the `ResError` (page 38) function returns the result code `resNotFound`. If the given index is larger than the value returned by `Count1Resources` (page 15), `Get1IndResource` (page 23) returns `NULL`, and `ResError` (page 38) returns the result code `resNotFound`. If the resource to be read won't fit into memory, the function returns `NULL`, and `ResError` (page 38) returns the appropriate result code.

**Discussion**
The function reads the resource data into memory if it's not already there, unless you've called the `SetResLoad` (page 41) function with the `load` parameter set to `FALSE`. If you've called `SetResLoad` with the `load` parameter set to `FALSE` and the data isn't already in memory, `Get1IndResource` (page 23) returns an empty handle (that is, a handle whose master pointer is set to `NULL`). This can also happen if you read resource data for a purgeable resource into memory and then call `SetResLoad` (page 41) with the `load` parameter set to `FALSE`. If the resource data is later purged and you call the `Get1IndResource` (page 23) function, the function 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` (page 35) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## Get1IndType

Gets a resource types available in the current resource file.

```
void Get1IndType (
   ResType *theType,
   ResourceIndex index
);
```

**Parameters**

*theType*

On return, the resource type with the specified index in the current resource file.

You can call this function 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`, this parameter contains four null characters (ASCII code 0).

*index*

An integer ranging from 1 to the number of resource types in the current resource file, as returned by the `Count1Types` (page 15) function.

**Discussion**
To check for errors, call the `ResError` (page 38) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## Get1NamedResource

Gets a named resource in the current resource file.

```
Handle Get1NamedResource (
   ResType theType,
   ConstStr255Param name
);
```

**Parameters**

*theType*

The resource type of the resource about which you wish to retrieve data.

*name*

A name that uniquely identifies the resource about which you wish to retrieve data.

**Return Value**

If the function 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 `NULL`, and if you haven't called the `SetResLoad` (page 41) function with the `load` parameter set to `FALSE`, `Get1NamedResource` attempts to read the resource into memory. If it can't find the resource data, the function returns `NULL`, and the `ResError` (page 38) function returns the result code `resNotFound`. The `Get1NamedResource` function also returns `NULL` 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 this function with a resource type that can't be found in the resource map of the current resource file, the function returns `NULL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NULL`.

**Discussion**

The function searches the current resource file's resource map in memory for the specified resource. You can change the search order by calling the `UseResFile` (page 46) function before `Get1NamedResource`.

**Availability**

Available in Mac OS X v10.0 and later.

**Declared In**

`Resources.h`

## Get1Resource

Gets resource data for a resource in the current resource file.

```
Handle Get1Resource (
   ResType theType,
   ResID theID
);
```

**Parameters**

*theType*

The resource type of the resource about which you wish to retrieve data.

*theID*

An integer that uniquely identifies the resource about which you wish to retrieve data.

**Return Value**

If the function 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 `NULL`, and if you haven't called the `SetResLoad` (page 41) function with the `load` parameter set to `FALSE`, `Get1Resource` attempts to read the resource into memory.

If the function can't find the resource data, it returns `NULL`, and `ResError` returns the result code `resNotFound`. The `Get1Resource` function also returns `NULL` 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 this function with a resource type that can't be found in the resource map of the current resource file, the function returns `NULL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NULL`.

**Discussion**
The function searches the current resource file's resource map in memory for the specified resource.

You can change the resource map search order by calling the `UseResFile` (page 46) function before `Get1Resource`.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
QTMetaData

**Declared In**
`Resources.h`

## GetIndResource

Returns a handle to a resource of a given type in resource forks open to your application.

```
Handle GetIndResource (
   ResType theType,
   ResourceIndex index
);
```

**Parameters**

*theType*
A resource type.

*index*
An integer ranging from 1 to the number of resources of a given type returned by the `CountResources` (page 16) function, which is the number of resources of that type in all resource forks open to your application.

**Return Value**
A handle to a resource of the given type. If you call this function 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 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. If you provide an index to that is either 0 or negative, the function returns `NULL`, and the `ResError` (page 38) function returns the result code `resNotFound`. If the given index is larger than the value returned by `CountResources`, the function returns `NULL`, and `ResError` (page 38) returns the result code `resNotFound`. If the resource to be read won't fit into memory, the function returns `NULL`, and `ResError` (page 38) returns the appropriate result code.

**Discussion**
This function reads the resource data into memory if it's not already there, unless you've called the `SetResLoad` (page 41) function with the `load` parameter set to `FALSE`.

If you've called SetResLoad (page 41) with the load parameter set to FALSE and the data isn't already in memory, the function returns an empty handle (a handle whose master pointer is set to NULL). 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, the function 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 (page 35) function.

The UseResFile (page 46) function 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 the Count1Resources (page 15) function and use the Get1IndResource (page 23) function to get handles to the resources of that type.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## GetIndType

Gets a resource type available in resource forks open to your application.

```
void GetIndType (
   ResType *theType,
   ResourceIndex index
);
```

**Parameters**

*theType*

On return, a pointer to the resource type for the specified index among all the resource forks open to your application.

You can call this function 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, this parameter contains four null characters (ASCII code 0).

*index*

An integer ranging from 1 to the number of resource types in all resource forks open to your application, as returned by CountTypes (page 16) function.

**Discussion**
To check for errors, call the ResError (page 38) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## GetMaxResourceSize

Returns the approximate size of a resource.

```
long GetMaxResourceSize (
   Handle theResource
);
```

### Parameters

*theResource*

A handle to the resource whose size you wish to retrieve.

### Return Value

The approximate size, in bytes, of the resource. Unlike the GetResourceSizeOnDisk (page 32) function, this function 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 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 the UpdateResFile (page 45) function before calling GetMaxResourceSize. If the handle isn't a handle to a valid resource, the function returns –1, and the ResError (page 38) function returns the result code resNotFound.

### Availability

Available in Mac OS X v10.0 and later.

### Declared In

Resources.h

## GetNamedResource

Gets a named resource.

```
Handle GetNamedResource (
   ResType theType,
   ConstStr255Param name
);
```

### Parameters

*theType*

The resource type of the resource about which you wish to retrieve data.

*name*

A name that uniquely identifies the resource about which you wish to retrieve data. Strings passed in this parameter are case-sensitive.

### Return Value

If the function 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 NULL, and if you haven't called the SetResLoad (page 41) function with the load parameter set to FALSE, GetNamedResource attempts to read the resource into memory.

If the function can't find the resource data, it returns NULL, and the ResError (page 38) function returns the result code resNotFound. The function also returns NULL 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 `NULL` as well, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NULL`.

**Discussion**
The function searches the resource maps in memory for the specified resource. The resource maps in memory, which represent all the open resource forks, are arranged as a linked list. When the function 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.

You can change the resource map search order by calling the `UseResFile` (page 46) function before `GetNamedResource`.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## GetNextFOND

Gets the next FOND handle.

```
Handle GetNextFOND (
   Handle fondHandle
);
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## GetNextResourceFile

Retrieves the next resource file in the resource chain.

```
OSErr GetNextResourceFile (
   ResFileRefNum curRefNum,
   ResFileRefNum *nextRefNum
);
```

**Parameters**

*curRefNum*
A value of type SInt16 representing the current reference number of a resource file.

*nextRefNum*
A pointer to a value of type SInt16. On return, this points to the next resource file in the resource chain.

**Return Value**
A result code. See "Resource Manager Result Codes" (page 56).

**Discussion**
`GetNextResourceFile` can be used to iterate over resource files in the resource chain. By passing a valid reference number in the `curRefNum` parameter, the function returns the reference number of the next file in the resource chain. If the resource file specified by the `curRefNum` parameter is not found in the resource chain, the `GetNextResourceFile` function returns the error code `resFNotFound`. When the end of the chain is reached `GetNextResourceFile` returns `noErr` and the value of the `nextRefNum` parameter is `NIL`.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## GetResAttrs

Gets a resource's attributes.

```
ResAttributes GetResAttrs (
   Handle theResource
);
```

**Parameters**

*`theResource`*

A handle to the resource whose attributes you wish to retrieve. If the value of this parameter isn't a handle to a valid resource, the function does nothing, and the `ResError` (page 38) function returns the result code `resNotFound`.

**Return Value**
The resource's attributes as recorded in its entry in the resource map in memory. The 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.

**Discussion**
To change a resource's attributes in the resource map in memory, use the `SetResAttrs` (page 39) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## GetResFileAttrs

Gets the attributes of a resource fork.

```
ResFileAttributes GetResFileAttrs (
   ResFileRefNum refNum
);
```

**Parameters**

*`refNum`*

A file reference number for the resource fork whose attributes you want to get. Specify 0 in this parameter to get the attributes of the System file's resource fork.

**Return Value**
The attributes of the file's resource fork. If there's no open resource fork for the given file reference number, the function does nothing, and the `ResError` (page 38) 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.

**Discussion**
The Resource Manager sets the `mapChanged` attribute for the resource fork when you call the `ChangedResource` (page 13), the `AddResource` (page 12), or the `RemoveResource` (page 37) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## GetResInfo

Gets a resource's resource ID, resource type, and resource name.

```
void GetResInfo (
   Handle theResource,
   ResID *theID,
   ResType *theType,
   Str255 name
);
```

**Parameters**

*theResource*
A handle to the resource for which you want to retrieve information. If the handle isn't a valid handle to a resource, the function does nothing to determine whether this has occurred, call the `ResError` (page 38) function.

*theID*
On return, a pointer to the resource ID of the specified resource.

*theType*
On return, a pointer to the resource type of the specified resource.

*name*
On return, the name of the specified resource.

**Discussion**
To set a resource's ID, resource type, or resource name, use the `SetResInfo` (page 40) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## GetResource

Gets resource data for a resource specified by resource type and resource ID.

```
Handle GetResource (
   ResType theType,
   ResID theID
);
```

**Parameters**

*theType*

The resource type of the resource about which you wish to retrieve data.

*theID*

An integer that uniquely identifies the resource about which you wish to retrieve data.

**Return Value**

If the 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 a handle whose value is `NULL`, and if you haven't called the `SetResLoad` (page 41) function with the `load` parameter set to `FALSE`, `GetResource` attempts to read the resource into memory.

If the function can't find the resource data, it returns `NULL`, and the `ResError` (page 38) function returns the result code `resNotFound`. The `GetResource` function also returns `NULL` 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 `NULL`, but `ResError` returns the result code `noErr`. You should always check that the value of the returned handle is not `NULL`.

**Discussion**

The function searches the resource maps in memory for the specified resource. The resource maps in memory, which represent all the open resource forks, are arranged as a linked list. When searching this list, the function 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.

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.

You can change the resource map search order by calling the `UseResFile` (page 46) function before calling `GetResource`.

**Availability**

Available in Mac OS X v10.0 and later.

**Related Sample Code**

Simple DrawSprocket

**Declared In**

`Resources.h`

## GetResourceSizeOnDisk

Returns the exact size of a resource.

```
long GetResourceSizeOnDisk (
   Handle theResource
);
```

**Parameters**

*theResource*

A handle to the resource whose size you wish to retrieve.

**Return Value**

The exact size, in bytes, of the resource. If the handle isn't a handle to a valid resource, the function returns –1, and the `ResError` (page 38) function returns the result code `resNotFound`.

**Discussion**

This function checks the resource on disk, not in memory. You can call this function before reading a resource into memory to make sure there's enough memory available to do so successfully.

The `GetResourceSizeOnDisk` function is also available as the `SizeResource` function.

**Availability**

Available in Mac OS X v10.0 and later.

**Declared In**

`Resources.h`

## GetTopResourceFile

Retrieves the topmost resource file in the current resource chain.

```
OSErr GetTopResourceFile (
   ResFileRefNum *refNum
);
```

**Parameters**

*refNum*

A pointer to a value of type `SInt16`. On return, this points to the top most resource file in the current resource chain.

**Return Value**

A result code. See "Resource Manager Result Codes" (page 56). If the resource chain is empty, `resFNotFound` is returned.

**Availability**

Available in Mac OS X v10.0 and later.

**Declared In**

`Resources.h`

## HomeResFile

Gets the file reference number associated with a particular resource.

```
ResFileRefNum HomeResFile (
   Handle theResource
);
```

**Parameters**

*theResource*

A handle to the resource for which you wish to get the associated file reference number.

**Return Value**

The file reference number for the resource fork containing the specified resource. If the given handle isn't a handle to a resource, the function returns –1, and the `ResError` (page 38) function returns the result code `resNotFound`. If `HomeResFile` returns 0, the resource is in the System file's resource fork. If it returns 1, the resource is ROM-resident.

**Availability**

Available in Mac OS X v10.0 and later.

**Declared In**

`Resources.h`

## InsertResourceFile

Inserts a resource file into the current resource chain at the specified location.

```
OSErr InsertResourceFile (
   ResFileRefNum refNum,
   RsrcChainLocation where
);
```

**Parameters**

*refNum*

A value of type SInt16 indicating the reference number of the resource file to insert into the resource chain.

*where*

A value of type RsrcChainLocation indicating where in the resource chain the resource file should be inserted. See the `RsrcChainLocation` data type.

**Return Value**

A result code. See "Resource Manager Result Codes" (page 56).

**Discussion**

If the file is already in the resource chain, it is removed and reinserted at the specified location. If the file has been detached, it is added to the resource chain at the specified location Returns resFNotFound if it's not currently open.

**Availability**

Available in Mac OS X v10.0 and later.

**Declared In**

`Resources.h`

## InvokeResErrUPP

Calls your callback function.

```
void InvokeResErrUPP (
    OSErr thErr,
    ResErrUPP userUPP
);
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## LoadResource

Gets resource data after you've called the SetResLoad function with the load parameter set to FALSE or when the resource is purgeable.

```
void LoadResource (
    Handle theResource
);
```

**Parameters**

*theResource*

A handle to a resource. Given this handle, the function reads the resource data into memory. If the resource is already in memory, or if the this parameter doesn't contain a handle to a resource, then the function does nothing. To determine whether either of these situations occurred, call the ResError (page 38) function. If the resource is already in memory, ResError returns noErr; if the handle is not a handle to a resource, ResError returns resNotFound.

**Discussion**
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, this function rereads the original resource from the file's resource fork. You should use the ChangedResource (page 13) or SetResPurge (page 43) function before calling LoadResource to ensure that changes made to purgeable resources are written to the resource fork.

**Availability**
Available in Mac OS X 10.0 and later.

**Declared In**
Resources.h

## NewResErrUPP

Creates a new universal procedure pointer (UPP) to your callback function.

```
ResErrUPP NewResErrUPP (
    ResErrProcPtr userRoutine
);
```

**Return Value**
See ResErrUPP (page 50) for more information.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## ReadPartialResource

Reads part of a resource into memory and work with a small subsection of a large resource.

```
void ReadPartialResource (
   Handle theResource,
   long offset,
   void *buffer,
   long count
);
```

**Parameters**

*theResource*
A handle to the resource you wish to read.

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

*buffer*
A pointer to the buffer into which the partial resource is to be read. Your application is responsible for the buffer's memory management. You cannot use the `ReleaseResource` (page 37) function to release the memory the buffer occupies.

*count*
The length of the resource subsection.

**Discussion**
This function 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` (page 38) 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`.

You may experience problems if you have a copy of a resource in memory when you are using the partial resource functions. If you have modified the copy in memory and then access the resource on disk using this function, the function reads the data on disk, not the data in memory, which is referenced through the resource's handle.

When using partial resource functions, you should call the `SetResLoad` (page 41) function, specifying `FALSE` for the `load` parameter, before you call `GetResource`. Using the `SetResLoad` function 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 the `GetResource` (page 31) function. Then use `ReadPartialResource` to read a portion of the resource into a buffer.

If the entire resource is in memory and you want only part of its data, it's faster to use the Memory Manager function `BlockMove` instead of the `ReadPartialResource` function. If you read a partial resource into memory and then change its size, you can use the `SetResourceSize` (page 42) function to change the entire resource's size on disk as necessary

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## ReleaseResource

Releases the memory a resource occupies when you have finished using it.

```
void ReleaseResource (
   Handle theResource
);
```

**Parameters**

*theResource*

A handle to the resource which you wish to release. The function sets the master pointer of the resource's handle in the resource map in memory to NULL. 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, the function does nothing, and the ResError (page 38) function 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**

Do not use this function to release a System resource that might be shared by several applications.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
Simple DrawSprocket

**Declared In**
Resources.h

## RemoveResource

Removes a resource's entry from the current resource file's resource map in memory.

```
void RemoveResource (
   Handle theResource
);
```

**Parameters**

*theResource*

A handle to the resource which you wish to detach. If the resProtected attribute for the resource is set or if this parameter doesn't contain a handle to a resource, the function does nothing, and the ResError (page 38) function returns the result code rmvResFailed.

**Discussion**
The RemoveResource function does not dispose of the handle you pass into it; to do so you must call the Memory Manager function DisposeHandle after calling RemoveResource. You should dispose the handle if you want to release the memory before updating or closing the resource fork.

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.

The `RemoveResource` function is also available as the `RmveResource` function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## ResError

Determines what error occurred, if any, after calling a Resource Manager function.

```
OSErr ResError (
   void
);
```

**Return Value**
A result code. See "Resource Manager Result Codes" (page 56). If no error occurred, the function returns `noErr`. If an error occurs at the Resource Manager level, the function returns one of the result codes specific to the Resource Manager. If an error occurs at the Operating System level, the function returns an Operating System result code. In certain cases, the `ResError` function returns `noErr` even though a Resource Manager function was unable to perform the requested operation. See the individual function descriptions for details about the circumstances under which this happens.

**Discussion**
Resource Manager functions do not report error information directly. Instead, after calling a Resource Manager function, your application should call this function to determine whether an error occurred. You also can use this function to check for an error after application startup (system software opens the resource fork of your application during application startup).

Resource Manager functions usually return `NULL` 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 `NULL`. If it is, your application can use this function 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 `NULL`.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
QTMetaData
Simple DrawSprocket

**Declared In**
`Resources.h`

## SetResAttrs

Sets a resource's attributes in the resource map in memory.

```
void SetResAttrs (
   Handle theResource,
   ResAttributes attrs
);
```

### Parameters

*theResource*

A handle to the resource whose attributes you wish to set. If the value of this parameter isn't a valid handle to a resource, the function does nothing, and the `ResError` (page 38) function returns the result code `resNotFound`.

*attrs*

The resource attributes to set. 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.

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.

### Discussion

This function changes the information in the resource map in memory, not in the file on disk. If you want the Resource Manager to write the modified resource map to disk after a subsequent call to the `UpdateResFile` (page 45) function or when your application terminates, call the `ChangedResource` (page 13) function after you call `SetResAttrs`.

Do not use this function to change a purgeable resource. If you make a purgeable resource nonpurgeable by setting the `resPurgeable` attribute with this function, 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.

You can check for errors using the `ResError` function. `SetResAttrs` 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 the `GetResAttrs` (page 30) function.

### Availability

Available in Mac OS X v10.0 and later.

### Declared In

`Resources.h`

## SetResFileAttrs

Sets a resource fork's attributes.

```
void SetResFileAttrs (
   ResFileRefNum refNum,
   ResFileAttributes attrs
);
```

**Parameters**

*refNum*

A file reference number for the resource fork whose attributes you want to set. If this value 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, the function does nothing, and the `ResError` (page 38) function returns the result code `noErr`.

*attrs*

The attributes to set. Like individual resources, resource forks have attributes that are specified by bits in the low-order byte of a word. When the Resource Manager first creates a resource fork after a call to `FSpOpenResFile` (page 60) or a related function, it does not set any of the resource fork's attributes—that is, they are all set to 0.

**Discussion**

The Resource Manager sets the `mapChanged` attribute for the resource fork when you call the `ChangedResource` (page 13), the `AddResource` (page 12), or the `RemoveResource` (page 37) function.

**Availability**

Available in Mac OS X v10.0 and later.

**Declared In**

`Resources.h`

## SetResInfo

Sets the name and resource ID of a resource.

```
void SetResInfo (
   Handle theResource,
   ResID theID,
   ConstStr255Param name
);
```

**Parameters**

*theResource*

A handle to the resource whose name and ID you wish to set.

*theID*

The new resource ID. If the parameter `theResource` doesn't contain a handle to an existing resource, the function does nothing, and the `ResError` (page 38) function returns the result code `resNotFound`.

*name*

The new name of the specified resource. If you pass an empty string for the `name` parameter, the resource name is not changed.

**Discussion**

The function changes the information in the resource map in memory, not in the resource file itself. 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 resource map becomes too large to fit in memory (for example, after an unnamed resource is given a name), this function 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` (page 13) function for the same resource after you call `SetResInfo`. Even if you don't call `ChangedResource` after using this function 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 the `UpdateResFile` (page 45) function 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.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## SetResLoad

Enables and disables automatic loading of resource data into memory for functions that return handles to resources.

```
void SetResLoad (
   Boolean load
);
```

**Parameters**

*load*

Determines whether Resource Manager functions should read resource data into memory. If you set this parameter to `TRUE`, Resource Manager functions 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 functions will not automatically read resource data into memory. Instead, such functions return a handle whose master pointer is set to `NULL` 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. The default setting is `TRUE`.

If you call the function with this parameter set to `FALSE`, be sure to call `SetResLoad` with this 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 functions won't work if resources are not loaded automatically.

**Discussion**
You can use the `SetResLoad` function 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 this function, call the `LoadResource` function.

To check for errors, call the `ResError` (page 38) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
Simple DrawSprocket

**Declared In**
`Resources.h`

## SetResourceSize

Sets the size of a resource on disk.

```
void SetResourceSize (
   Handle theResource,
   long newSize
);
```

**Parameters**

*theResource*
A handle to the resource which you wish to change.

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

**Discussion**
This function is normally used only with the `ReadPartialResource` (page 36) and `WritePartialResource` (page 47) functions.

This function 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 you read a partial resource into memory and then change its size, you must use this function 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 this function. For instance, suppose you read an entire resource into memory by calling `GetResource` (page 31) or related functions, then use `SetResourceSize` successfully to set the resource size on disk, and finally attempt to write the resource to disk using the `UpdateResFile` (page 45) or `WriteResource` (page 48) functions. In this case, the Resource Manager adjusts the resource size on disk to conform with the size of the resource in memory.

If the disk is locked or full, or the file is locked, this function does nothing, and the `ResError` (page 38) 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.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## SetResPurge

Tells the Memory Manager to pass the handle of a resource to the Resource Manager before purging the data specified by that handle.

```
void SetResPurge (
   Boolean install
);
```

**Parameters**

*install*

Specifies whether the Memory Manager checks with the Resource Manager before purging a resource handle.

Specify TRUE 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 (page 48) function to write the resource's resource data to the resource fork before returning control to the Memory Manager.

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

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

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

**Discussion**
You can use this function 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.

To check for errors, call the ResError (page 38) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## Unique1ID

Gets a resource ID that's unique with respect to resources in the current resource file.

```
ResID Unique1ID (
   ResType theType
);
```

**Parameters**

*theType*

A resource type.

**Return Value**

A resource ID greater than 0 that isn't currently assigned to any resource of the specified type in the current resource file.

**Discussion**

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.

To check for errors, call the `ResError` (page 38) function.

For more information about restrictions on resource IDs for specific resource types, see `ResID` (page 50).

In versions of system software earlier than System 7, this 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 `Unique1ID` again, and continue doing so until you get a resource ID greater than 127.

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

**Availability**

Available in Mac OS X v10.0 and later.

**Declared In**

`Resources.h`

## UniqueID

Gets a unique resource ID for a resource.

```
ResID UniqueID (
   ResType theType
);
```

**Parameters**

*theType*

A resource type.

**Return Value**

A resource ID greater than 0 that isn't currently assigned to any resource of the specified type in any open resource fork.

**Discussion**

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.

To check for errors, call the `ResError` (page 38) function.

For more information about restrictions on resource IDs for specific resource types, see `ResID` (page 50).

In versions of system software earlier than System 7, this 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.

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

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## UpdateResFile

Updates the resource map and resource data for a resource fork without closing it.

```
void UpdateResFile (
   ResFileRefNum refNum
);
```

**Parameters**

*refNum*

A file reference number for a resource fork. If there's no open resource fork with the given reference number, the function does nothing, and the `ResError` (page 38) 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 this function but the `mapReadOnly` attribute of the resource fork is set, the function does nothing, and the `ResError` function returns the result code `resAttrErr`.

**Discussion**
Given the reference number of a file whose resource fork is open, this function 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 the `ChangedResource` (page 13) or `AddResource` (page 12) function. The `UpdateResFile` function calls the `WriteResource` (page 48) function 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` function 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`.

Because the `CloseResFile` (page 14) function 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.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

## UseResFile

Sets the current resource file.

```
void UseResFile (
   ResFileRefNum refNum
);
```

**Parameters**

*refNum*

The file reference number for the resource fork which you wish to set as the current resource fork.

**Return Value**
The function searches the list of files whose resource forks have been opened for the file specified here. 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, the function does nothing. To set the current resource file to the System file, use 0 here.

**Discussion**
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 this function 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 this function 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.

You typically call the `CurResFile` (page 16) function 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.

Most of the Resource Manager functions 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.

The `FSpOpenResFile` (page 60) and `HOpenResFile` (page 63) functions, which also set the current resource file, override previous calls to `UseResFile`.

To check for errors, call the `ResError` (page 38) function.

**Availability**
Available in Mac OS X v10.0 and later.

**Related Sample Code**
Simple DrawSprocket

**Declared In**
`Resources.h`

## WritePartialResource

Writes part of a resource to disk when working with a small subsection of a large resource.

```
void WritePartialResource (
   Handle theResource,
   long offset,
   const void *buffer,
   long count
);
```

**Parameters**

*theResource*
A handle to the resource you wish to write to disk.

*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. Your application is responsible for the buffer's memory management.

*count*
The length of the resource subsection to write.

**Discussion**
If the disk or the file is locked, the `ResError` (page 38) 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`.

This function 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 functions, you should call the `SetResLoad` (page 41) function, specifying `FALSE` for the `load` parameter, before you call the `GetResource` (page 31) function. 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 the `SetResourceSize` (page 42) function to change the entire resource's size on disk as necessary before you write the partial resource.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## WriteResource

Writes resource data in memory immediately to a file's resource fork.

```
void WriteResource (
   Handle theResource
);
```

**Parameters**

*theResource*

A handle to a resource. The function checks the resChanged attribute of this resource. If the resChanged attribute is set to 1, such as after a successful call to the ChangedResource (page 13) or AddResource (page 12) function, WriteResource writes the resource data in memory to the resource fork, then clears the resChanged attribute in the resource's resource map in memory.

If the resource is purgeable and has been purged, the function writes zero-length resource data to the resource fork. If the resource's resProtected attribute is set to 1, the function does nothing, and the ResError (page 38) 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.

**Discussion**
Note that this function does not write the resource's resource map entry to disk.

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 function 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.

The resource fork is updated automatically when your application quits, when you call the UpdateResFile (page 45) function, or when you call the CloseResFile (page 14) function. Thus, you should call WriteResource only if you want to write just one or a few resources immediately.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

# Callbacks

### ResErrProcPtr

```
typedef void (*ResErrProcPtr) (
    OSErr thErr
);
```

If you name your function `MyResErrProc`, you would declare it like this:

```
void MyResErrProc (
    OSErr thErr
);
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

### ResourceEndianFilterPtr

```
typedef OSErr (*ResourceEndianFilterPtr) (
    Handle theResource,
    Boolean currentlyNativeEndian
);
```

If you name your function `MyResourceEndianFilter`, you would declare it like this:

```
OSErr MyResourceEndianFilter (
    Handle theResource,
    Boolean currentlyNativeEndian
);
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

# Data Types

### ResAttributes

```
typedef short ResAttributes;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## ResErrUPP

```
typedef ResErrProcPtr ResErrUPP;
```

**Discussion**
For more information, see the description of the ResErrProcPtr (page 49) callback function.

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## ResFileAttributes

```
typedef short ResFileAttributes;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## ResFileRefNum

```
typedef short ResFileRefNum;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
Resources.h

## ResID

Defines a unique identifier for a resource of a given type.

```
typedef short ResID;
```

**Discussion**
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 functions and font families, fall in different ranges or in ranges that are broken down for more specific purposes.

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).

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`Resources.h`

### ResType

Defines a unique identifier for a type of resource.

```
typedef FourCharCode ResType;
```

**Discussion**
The Resource Manager uses the resource type along with the resource ID to identify a resource. 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 not conflict with any of the standard resource types. When identifying resource types, the Resource Manager distinguishes between uppercase letters and their lowercase counterparts. 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).

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`IOMacOSTypes.h`

## Constants

### Reference Number Constants

```
enum {
    kResFileNotOpened = -1,
    kSystemResFile = 0
};
```

**Constants**
`kResFileNotOpened`
Indicates the reference number returned as error when opening a resource file.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`kSystemResFile`
Indicates the default reference number to the system file.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

## Resource Attribute Bits

```
enum {
    resSysRefBit = 7,
    resSysHeapBit = 6,
    resPurgeableBit = 5,
    resLockedBit = 4,
    resProtectedBit = 3,
    resPreloadBit = 2,
    resChangedBit = 1,
};
```

**Constants**

`resSysRefBit`
If this attribute is set to 1, it is a system reference. If it is set to 0, it is a local reference.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

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

If you are setting your resource's attributes with `SetResAttrs`, you should set this bit to 0 for your application's resources. Note that if you do set the `resSysHeapBit` 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.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`resPurgeableBit`
If this attribute is set to 1, the resource is purgeable if it's 0, the resource is nonpurgeable. However, do not use `SetResAttrs` to make a purgeable resource nonpurgeable.

Because a locked resource is nonrelocatable and nonpurgeable, the `resLockedBit` attribute overrides the `resPurgeableBit` attribute.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`resLockedBit`
If this attribute is 1, the resource is nonpurgeable regardless of whether `resPurgeableBit` is set. If it's 0, the resource is purgeable or nonpurgeable depending on the value of the `resPurgeableBit` attribute.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`resProtectedBit`
: If this attribute is set to 1, your application can't use Resource Manager functions 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` function to remove this protection. Note that this attribute change takes effect immediately.

  Available in Mac OS X v10.0 and later.

  Declared in `Resources.h`.

`resPreloadBit`
: If this 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 `resPreloadBit` attribute and the `resLockedBit` attribute are set, the Resource Manager loads the resource as low in the heap as possible.

  Available in Mac OS X v10.0 and later.

  Declared in `Resources.h`.

`resChangedBit`
: If this 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 `resChangedBit` attribute must be 0 in the resource fork on disk.

  Do not use `SetResAttrs` to set the `resChangedBit` attribute. Be sure the `attrs` parameter passed to `SetResAttrs` doesn't change the current setting of this attribute. To set the `resChangedBit` attribute, call the `ChangedResource` function.

  Available in Mac OS X v10.0 and later.

  Declared in `Resources.h`.

**Discussion**
The `SetResAttrs` (page 39) and `GetResAttrs` (page 30) functions use these constants to refer to each attribute.

## Resource Attribute Masks

```
enum {
    resSysHeap = 64,
    resPurgeable = 32,
    resLocked = 16,
    resProtected = 8,
    resPreload = 4,
    resChanged = 2,
};
```

**Constants**
`resSysHeap`
: Use to set or test for the `resSysHeapBit`.

  Available in Mac OS X v10.0 and later.

  Declared in `Resources.h`.

`resPurgeable`
: Use to set or test for the `resPurgeableBit`.

  Available in Mac OS X v10.0 and later.

  Declared in `Resources.h`.

`resLocked`
Use to set or test for the `resLockedBit`.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`resProtected`
Use to set or test for the `resProtectedBit`.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`resPreload`
Use to set or test for the `resPreloadBit`.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`resChanged`
Use to set or test for the `resChangedBit`.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

## Resource Chain Location

Specify the location of the resource chain.

```
typedef SInt16 RsrcChainLocation
enum {
    kRsrcChainBelowSystemMap = 0,
    kRsrcChainBelowApplicationMap = 1,
    kRsrcChainAboveApplicationMap = 2,
    kRsrcChainAboveAllMaps = 4
};
```

**Constants**

`kRsrcChainBelowSystemMap`
Indicates the resource chain is below the system's resource map.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`kRsrcChainBelowApplicationMap`
Indicates the resource chain is below the application's resource map.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`kRsrcChainAboveApplicationMap`
Indicates the resource chain is above the application's resource map.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`kRsrcChainAboveAllMaps`
Indicates the resource chain is above all resource maps.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

**Discussion**
These constants and data type are for use with the Resource Manager chain manipulation routines under Carbon.

## Resource Fork Attribute Bits

```
enum {
    mapReadOnlyBit = 7,
    mapCompactBit = 6,
    mapChangedBit = 5
};
```

**Constants**

`mapReadOnlyBit`

If this bit 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 `ChangedResource` (page 13) and `WriteResource` (page 48) functions do nothing, but the function `ResError` (page 38) returns the result code `noErr`.

If you set the `mapReadOnlyBit` 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.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`mapCompactBit`

If this bit 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 `mapCompactBit` attribute to force the Resource Manager to compact a resource fork when your changes have made resources smaller.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`mapChangedBit`

If this bit is set to 1, the Resource Manager writes the resource map to disk when the file is updated. For example, you can set `mapChangedBit` if you've changed resource attributes only and don't want to call the `ChangedResource` (page 13) function because you don't want to write the resource data to disk.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

### Resource Fork Attribute Masks

```
enum{
    mapReadOnly = 128,
    mapCompact = 64,
    mapChanged = 32
};
```

**Constants**

`mapReadOnly`

Use to set or test for the `mapReadOnlyBit`.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`mapCompact`

Use to set or test for the `mapCompactBit`.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

`mapChanged`

Use to set or test for the `mapChangedBit`.

Available in Mac OS X v10.0 and later.

Declared in `Resources.h`.

## Result Codes

The most common result codes returned by Resource Manager are listed in the table below. The Resource Manager may also return the following result codes: `noErr` (0), `dirFulErr` (-33), `dskFulErr` (-34), `nsvErr` (-35), `ioErr` (-36), `bdNamErr` (-37), `eofErr` (-39), `tmfoErr` (-42), `fnfErr` (-43), `wPrErr` (-44), `fLckdErr` (-45), `vLckdErr` (-46), `dupFNErr` (-48), `opWrErr` (-49), `permErr` (-54), `extFSErr` (-58), `memFullErr` (-108), `dirNFErr` (-120).

| Result Code | Value | Description |
|---|---|---|
| `badExtResource` | -185 | The extended resource has a bad format.<br>Available in Mac OS X v10.0 and later. |
| `CantDecompress` | -186 | Can't decompress a compressed resource.<br>Available in Mac OS X v10.0 and later. |
| `resourceInMemory` | -188 | The resource is already in memory.<br>Available in Mac OS X v10.0 and later. |
| `writingPastEnd` | -189 | Writing past the end of file.<br>Available in Mac OS X v10.0 and later. |
| `inputOutOfBounds` | -190 | The offset or count is out of bounds.<br>Available in Mac OS X v10.0 and later. |

| Result Code | Value | Description |
| --- | --- | --- |
| `resNotFound` | -192 | The resource was not found.<br>Available in Mac OS X v10.0 and later. |
| `resFNotFound` | -193 | The resource file was not found.<br>Available in Mac OS X v10.0 and later. |
| `addResFailed` | -194 | The `AddResource` function failed.<br>Available in Mac OS X v10.0 and later. |
| `rmvResFailed` | -196 | The `RemoveResource` function failed.<br>Available in Mac OS X v10.0 and later. |
| `resAttrErr` | -198 | The attribute is inconsistent with the operation.<br>Available in Mac OS X v10.0 and later. |
| `mapReadErr` | -199 | The map is inconsistent with the operation.<br>Available in Mac OS X v10.0 and later. |

# Deprecated Resource Manager Functions

A function identified as deprecated has been superseded and may become unsupported in the future.

## Deprecated in Mac OS X v10.5

### FSpCreateResFile

Creates an empty resource fork in a new or existing file. (Deprecated in Mac OS X v10.5. Use `FSCreateResourceFile` (page 19) instead.)

```
void FSpCreateResFile (
   const FSSpec *spec,
   OSType creator,
   OSType fileType,
   ScriptCode scriptTag
);
```

**Discussion**
This function is not recommended. You should use a file's data fork instead of its resource fork to store resource data.

**Availability**
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.5.
Not available to 64-bit applications.

**Declared In**
`Resources.h`

### FSpOpenOrphanResFile

Opens a resource file that is persistent across all contexts. (Deprecated in Mac OS X v10.5. Use `FSOpenOrphanResFile` (page 21) instead.)

```
OSErr FSpOpenOrphanResFile (
   const FSSpec *spec,
   SignedByte permission,
   ResFileRefNum *refNum
);
```

**Return Value**
A result code. See "Resource Manager Result Codes" (page 56).

**Discussion**
`FSpOpenOrphanResFile` should be used to open a resource file that is persistent across all contexts. `FSpOpenOrphanResFile` loads everything into the system context and detaches the file from the context in which it was opened. If the file is already in the resource chain and a new instance is not opened, `FSpOpenOrphanResFile` will return a `paramErr`. Use this function with care, as it can and will fail if the map is very large or a lot of resources are preloaded.

**Availability**
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.5.
Not available to 64-bit applications.

**Declared In**
`Resources.h`

## FSpOpenResFile

Opens the resource fork in a file specified with an `FSSpec` structure. (Deprecated in Mac OS X v10.5. Use `FSOpenResourceFile` (page 22) instead.)

```
ResFileRefNum FSpOpenResFile (
   const FSSpec *spec,
   SignedByte permission
);
```

**Parameters**

*spec*
A pointer to a file system specification record specifying the name and location of the file whose resource fork is to be opened. This function also makes the specified file the current resource file.

*permission*
A constant indicating the type of access with which to open the resource fork. For a description of the types of access you can request, see File Access Permission Constants in *File Manager Reference*.

**Return Value**
The file reference number for the resource fork. 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 functions.

If you attempt to use this function to open a resource fork that is already open, it 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, the function doesn't make that file the current resource file.

If the 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` (page 38) function to determine what kind of error occurred.

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`.

**Discussion**

This function is available only in System 7 and later versions of system software. You can determine whether `FSpOpenResFile` is available by calling the `Gestalt` function with the `gestaltFSAttr` selector code. If this function is not available to your application, you can use `HOpenResFile`, `OpenRFPerm`, or `OpenResFile` instead. The `HOpenResFile` (page 63) function is preferred if `FSpOpenResFile` is not available. The `OpenRFPerm` (page 64) function is an earlier version of `HOpenResFile` that is still supported but is more restricted in its capabilities.

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.

You don't have to call this function 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` (page 16) 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`.

It's possible to create multiple, unique, read-only access paths to a resource fork using this function 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 this function immediately before your application reads information from the file and close the file immediately afterward. Otherwise, your application may get unexpected results.

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 this function with calls to the `SetResLoad` (page 41) function 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 the `LoadSeg` function 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 the `UseResFile` (page 46) function with your application's file reference number to make your application the current resource file.

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**

Because there is no support for locking and unlocking file ranges on local disks in Mac OS X, regardless of whether File Sharing is enabled, you cannot open more than one path to a resource fork with read/ write permission. If you try to open a more than one path to a file's resource fork with `fsRdWrShPerm` permission, only the first attempt will succeed. Subsequent attempts will return an invalid reference number and the `ResError` function will return the error `opWrErr`.

**Availability**

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.5.

Not available to 64-bit applications.

**Related Sample Code**
Simple DrawSprocket

**Declared In**
`Resources.h`

## FSpResourceFileAlreadyOpen

Checks whether a resource file is open. (Deprecated in Mac OS X v10.5. Use `FSResourceFileAlreadyOpen` (page 22) instead.)

```
Boolean FSpResourceFileAlreadyOpen (
   const FSSpec *resourceFile,
   Boolean *inChain,
   ResFileRefNum *refNum
);
```

**Parameters**

*resourceFile*

The resource file to check.

*inChain*

A pointer to a variable allocated by the caller. On return, `true` if the resource file is in the resource chain, `false` otherwise.

*refNum*

A pointer to a variable allocated by the caller. On return, the reference number of the file if it is open.

**Return Value**
This function returns `true` if the resource file is already open and known by the Resource Manager—for example, it is either in the current resource chain or it is a detached resource file.

**Availability**
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.5.
Not available to 64-bit applications.

**Declared In**
`Resources.h`

## HCreateResFile

Creates an empty resource fork, when the `FSpCreateResFile` function is not available. (Deprecated in Mac OS X v10.5. Use `FSCreateResourceFile` (page 19) instead.)

```
void HCreateResFile (
   FSVolumeRefNum vRefNum,
   long dirID,
   ConstStr255Param fileName
);
```

**Discussion**
This function is not recommended. You should use a file's data fork instead of its resource fork to store resource data.

**Availability**
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.5.
Not available to 64-bit applications.

**Declared In**
`Resources.h`

## HOpenResFile

Opens a file's resource fork, when the `FSpOpenResFile` function is not available. (Deprecated in Mac OS X v10.5. Use `FSOpenResourceFile` (page 22) instead.)

```
ResFileRefNum HOpenResFile (
   FSVolumeRefNum vRefNum,
   long dirID,
   ConstStr255Param fileName,
   SInt8 permission
);
```

**Parameters**

*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 indicating the type of access with which to open the resource fork. For a description of the types of access you can request, see File Access Permission Constants in *File Manager Reference*.

**Return Value**
The file reference number for the file. You can use this file reference number to refer to the file in other Resource Manager functions. The function also makes this file the current resource file. If the file's resource fork is already open, the function returns the file reference number but does not make that file the current resource file. If the 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` (page 38) function to determine what kind of error occurred.

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

**Discussion**
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.

You don't have to call this function 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` (page 16) 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. It's possible to create multiple, unique, read-only access paths to a resource fork using HOpenResFile; however, you should avoid doing so, to prevent inconsistencies between multiple copies of the resource map. See the discussion of this issue in relation to FSpOpenResFile (page 60). The HOpenResFile function works the same way.

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.

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 the SetResLoad (page 41) function 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 60) also applies to HOpenResFile.

**Special Considerations**

Because there is no support for locking and unlocking file ranges on local disks in Mac OS X, regardless of whether File Sharing is enabled, you cannot open more than one path to a resource fork with read/write permission. If you try to open a more than one path to a file's resource fork with fsRdWrShPerm permission, only the first attempt will succeed. Subsequent attempts will return an invalid reference number and the ResError function will return the error opWrErr.

**Availability**

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.5.

Not available to 64-bit applications.

**Declared In**

Resources.h

## OpenRFPerm

Opens a file's resource fork, when the FSpOpenResFile and HOpenResFile functions are not available. (Deprecated in Mac OS X v10.5. Use FSOpenResourceFile (page 22) instead.)

```
ResFileRefNum OpenRFPerm (
   ConstStr255Param fileName,
   FSVolumeRefNum vRefNum,
   SInt8 permission
);
```

**Parameters**

*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 indicating the type of access with which to open the resource fork. For a description of the types of access you can request, see File Access Permission Constants in *File Manager Reference*.

**Return Value**
The 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 functions. The function also makes this file the current resource file. If the file's resource fork is already open, the function returns the file reference number but does not make that file the current resource file.

If the 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` (page 38) function to determine what kind of error occurred.

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

**Discussion**
You can use this function if the `FSpOpenResFile` (page 60) function is not available. You can determine whether `FSpOpenResFile` is available by calling the `Gestalt` function with the `gestaltFSAttr` selector code. 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 `OpenRFPerm` is an earlier versions of `HOpenResFile` that is still supported but is more restricted in its capabilities.

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.

You don't have to call this function 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` (page 16) function after the application starts up and before you open the resource forks for any other files.

This function checks that the information in the resource map is internally consistent. If it isn't, `ResError` returns the result code `mapReadErr`. It's possible to create multiple, unique, read-only access paths to a resource fork using this function however, you should avoid doing so, to prevent inconsistencies between multiple copies of the resource map.

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`.

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 this function with calls to the `SetResLoad` (page 41) function 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.

**Availability**
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.5.
Not available to 64-bit applications.

**Declared In**
`Resources.h`

# Document Revision History

This table describes the changes to *Resource Manager Reference*.

| Date | Notes |
| --- | --- |
| 2007-10-31 | Moved functions out of the Miscellaneous section. Added information about the ResType data type. |
| 2006-09-13 | Updated for Mac OS X v10.5. |
| 2005-04-29 | Added new information about file permissions. |
| 2003-02-01 | Added documentation for the functions `FSCreateResourceFile`, `FSCreateResourceFork`, `FSOpenResourceFile`, `FSpResourceFileAlreadyOpen`, `GetNextResourceFile`, and `GetTopResourceFile`. |
| | Updated the constants section. |
| 2001-07-01 | First version of this document. |

Document Revision History

# Index


## A

addResFailed constant 57
AddResource function 12

## B

badExtResource constant 56

## C

CantDecompress constant 56
ChangedResource function 13
CloseResFile function 14
Count1Resources function 15
Count1Types function 15
CountResources function 16
CountTypes function 16
CurResFile function 16

## D

DetachResource function 17
DetachResourceFile function 18
DisposeResErrUPP function 18

## F

FSCreateResFile function 18
FSCreateResourceFile function 19
FSCreateResourceFork function 20
FSOpenOrphanResFile function 21
FSOpenResFile function 21
FSOpenResourceFile function 22
FSpCreateResFile function (Deprecated in Mac OS X v10.5) 59
FSpOpenOrphanResFile function (Deprecated in Mac OS X v10.5) 59
FSpOpenResFile function (Deprecated in Mac OS X v10.5) 60
FSpResourceFileAlreadyOpen function (Deprecated in Mac OS X v10.5) 62
FSResourceFileAlreadyOpen function 22

## G

Get1IndResource function 23
Get1IndType function 24
Get1NamedResource function 24
Get1Resource function 25
GetIndResource function 26
GetIndType function 27
GetMaxResourceSize function 28
GetNamedResource function 28
GetNextFOND function 29
GetNextResourceFile function 29
GetResAttrs function 30
GetResFileAttrs function 30
GetResInfo function 31
GetResource function 31
GetResourceSizeOnDisk function 32
GetTopResourceFile function 33

## H

HCreateResFile function (Deprecated in Mac OS X v10.5) 62
HomeResFile function 33
HOpenResFile function (Deprecated in Mac OS X v10.5) 63

## I

inputOutOfBounds constant 56
InsertResourceFile function 34
InvokeResErrUPP function 34

## K

kResFileNotOpened constant 51
kRsrcChainAboveAllMaps constant 54
kRsrcChainAboveApplicationMap constant 54
kRsrcChainBelowApplicationMap constant 54
kRsrcChainBelowSystemMap constant 54
kSystemResFile constant 52

## L

LoadResource function 35

## M

mapChanged constant 56
mapChangedBit constant 55
mapCompact constant 56
mapCompactBit constant 55
mapReadErr constant 57
mapReadOnly constant 56
mapReadOnlyBit constant 55

## N

NewResErrUPP function 35

## O

OpenRFPerm function (Deprecated in Mac OS X v10.5) 64

## R

ReadPartialResource function 36
Reference Number Constants 51
ReleaseResource function 37
RemoveResource function 37
resAttrErr constant 57
ResAttributes data type 49
resChanged constant 54
resChangedBit constant 53
ResError function 38
ResErrProcPtr callback 49
ResErrUPP data type 50
ResFileAttributes data type 50
ResFileRefNum data type 50
resFNotFound constant 57
ResID data type 50
resLocked constant 54
resLockedBit constant 52
resNotFound constant 57
Resource Attribute Bits 52
Resource Attribute Masks 53
Resource Chain Location 54
Resource Fork Attribute Bits 55
Resource Fork Attribute Masks 56
ResourceEndianFilterPtr callback 49
resourceInMemory constant 56
resPreload constant 54
resPreloadBit constant 53
resProtected constant 54
resProtectedBit constant 53
resPurgeable constant 53
resPurgeableBit constant 52
resSysHeap constant 53
resSysHeapBit constant 52
resSysRefBit constant 52
ResType data type 51
rmvResFailed constant 57

## S

SetResAttrs function 39
SetResFileAttrs function 39
SetResInfo function 40
SetResLoad function 41
SetResourceSize function 42
SetResPurge function 43

## U

Unique1ID function 43
UniqueID function 44
UpdateResFile function 45
UseResFile function 46

## W

WritePartialResource function 47
WriteResource function 48
writingPastEnd constant 56