INSIDE MACINTOSH

File Navigation and Access

WWDC Release

May 1996

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

Simultaneously published in the United States and Canada.

Contents

Figures

File Navigation and Access Overview

Contents

This chapter introduces the file navigation and access components of the new files environment available with Mac OS 8. These components support the improved user interface available with Mac OS 8 and provide many new capabilities. The components include

- Mac OS 8 File Manager
- Navigation Services
- Folder Manager
- Alias Manager
- Standard C I/O
- Translation Manager

You should read this chapter if your product manages or manipulates files. If your software product creates, opens, closes, saves, or renames files, it can take advantage of the Mac OS 8 File Manager features for improved performance. If your product provides access to volume formats other than HFS (for example, a standard format such as DOS FAT or a custom format optimized for your customers' particular needs), the Mac OS 8 volume-format plug-ins simplify your product development.

This chapter briefly describes the features of the files environment components and introduces aspects of the Mac OS 8 files architecture.

The Mac OS 8 Files Environment

This section describes the features of the Mac OS 8 application programming interfaces to the File Manager and the benefits of the new functions in other Mac OS 8 files environment components to users and developers.

The Mac OS 8 File Manager

The Mac OS 8 File Manager application programming interface contains about half as many functions as the System 7 File Manager, yet it provides a more powerful interface. The Mac OS 8 File Manager is designed to

- run fast and efficiently

- be fully reentrant
- be concurrent; that is, be able to run multiple requests simultaneously
- support a variety of volume formats
- make extensive use of the Mac OS 8 event notification system
- support international filenames by using text objects as well as text strings
- work closely with the virtual memory system and microkernel

Speed

The Mac OS 8 File Manager provides significant performance improvements over previous versions of the File Manager.

- **Increased efficiency.** Better algorithms improve the performance of the Mac OS 8 File Manager. For example, the Mac OS 8 File Manager can perform several operations concurrently, and it dispatches and schedules many threads of execution to complete its work as quickly as possible.
- **High-performance paging and memory cache.** The Mac OS 8 File Manager is integrated with the Mac OS 8 virtual memory system to provide high-performance paging and a new cache architecture, both of which take advantage of the I/O subsystem in Mac OS 8. This integration results in dramatic improvements in memory use.
- **Multitasking.** By taking advantage of the Mac OS 8 multitasking capabilities, your application can create secondary tasks to efficiently perform file I/O processing in the background.

Reentrancy

The Mac OS 8 files environment is fully reentrant. You can call the File Manager from any task; you are not restricted to calling it from main tasks of cooperative programs.

Support for Volume-Format Plug-ins

The Mac OS 8 File Manager does not include code that is specific to a particular volume format. Instead, it dispatches messages to plug-ins that handle the I/O for a specific volume format. To facilitate support for different volume types,

Apple Computer provides several plug-ins for volume formats. The Mac OS 8 File Manager supports

- HFS
- AppleTalk Filing Protocol (AFP), which allows access to AppleShare and Personal FileShare volumes
- DOS FAT
- These common CD-ROM formats:
  - High Sierra
  - ISO 9960
  - Photo CD
  - Audio CD
- Other third-party formats, such as Novell's NetWare

Use of Notification Services

Mac OS 8 provides an event notification system that allows code modules to publish the occurrence of particular events. Other code modules can subscribe to be notified when specific events occur. Notification of events may be exchanged among the following files-related code modules:

- File Manager
- volume-format plug-ins
- clients of the File Manager (applications, servers, and so forth)
- the block storage, SCSI, and other I/O families

International String Support

The Mac OS 8 File Manager uses text objects in addition to Pascal and C text strings to handle volume, folder, and file names and comments. Text objects include information about the language system, the text encoding system used, and the character codes used to represent text. Text objects can use any system of character codes, including Unicode. The use of text objects allows the File Manager to properly interpret file, folder, and volume names in any 1-byte or 2-byte language.

Virtual Memory and Microkernel Integration

The File Manager and virtual memory system in Mac OS 8 are closely integrated to provide highly efficient virtual memory paging.

Similarly, the microkernel works directly with the File Manager to clean up after the unexpected termination of a process or other exceptional condition.

File Manager Tasks

The Mac OS 8 File Manager manages the organization, reading, and writing of data located on persistent media. To do so, the File Manager can perform tasks such as those listed here:

- creating, moving, and renaming file and folder objects
- opening and closing files
- getting and setting simple properties
- using stream or memory-mapped file access methods to get and set fork properties
- copying files and directories
- establishing a position in an opened file
- allocating and deallocating storage
- obtaining information about specific file, folder, or volume objects
- iterating over file system objects such as files and folders to get information about them or to perform operations on them
- changing the scope of an iteration and directing the iteration's movement through the objects in folders or volumes
- manipulating memory-mapped files
- resolving pathnames and resolving object references for a given volume

Navigation Services

The Mac OS 8 Navigation Services provides an improved user experience that is consistent with other features of the Mac OS 8 human interface. Although calls to the Standard File Package continue to work in Mac OS 8, Navigation

Services provides a better and more easily customizable browsing interface and better access to document-specific information from within an application.

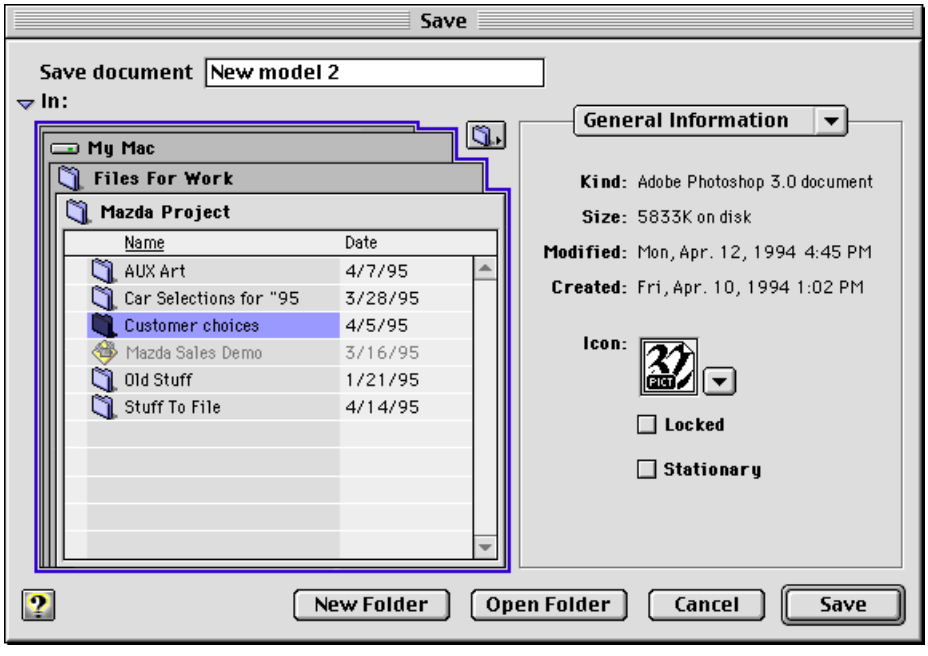
Here are some of the improvements in the Mac OS 8 human interface that make document management easier for the user:

- In Mac OS 8, when a user chooses the New command from the File menu, the new document is saved to disk when it is first created. Users can set a default location for new documents using the Default Location control panel. The first time they choose the Save command for a new untitled document, users are given an opportunity to change its name and location.
- In Mac OS 8, the Save As command in the File menu becomes the Save A Copy command. The behavior of the Save A Copy command is more readily understood by users.
- The Find Document command gives users access to the Mac OS 8 Finder's improved Find facility from within an application.
- The Open, Save, and Save A Copy commands use an intuitive navigation browser that you can customize for your application. The Navigation Browser makes it easy for users to access favorite items quickly, create new folders, and perform other common tasks that are not supported by the Standard File Package.
- The Document Info command in the Edit menu invokes a document information panel, similar to the current Get Info window in the Finder, that allows the user to manipulate document-specific information without switching contexts. The same document information panel also appears in the Open and Save dialog boxes. These panels also support Open Doc parts, providing a special set of Open Doc–specific subpanels.
- Users can define their "favorite items" list, which provides a visual menu of choices. These items can include servers, folders, applications, and files. There is also improved rebounding to allow users to access the last folder used and other recently selected items.
- There are alternative Save dialog boxes. There is a simple dialog box that saves a document by using default location settings or by allowing a user to select a favorite item as a location. The customized expanded Save dialog box allows users much greater flexibility and provides a degree of progressive disclosure as the user begins to make choices.

Figure 1-1 shows the Navigation Browser and one of several document information panels as they appear in the Save dialog box. You can easily

specify filters for the files displayed and let the user choose among available filters with a pop-up menu at the bottom of the Navigation Browser. You can also add specialized document information panels for your application's documents that permit editing and display of document information such as file type, author, keywords, colors, and dimensions.

Figure 1-1 Standard Navigation Services Save dialog box



Navigation Services also makes it much easier for applications to display an Open or Save dialog box with just one call, specifying options by using parameters rather than separate calls.

The Translation Manager

The Translation Manager's services allow your application to open documents created by other applications (possibly running on other operating systems)

and to import data from other applications with better fidelity than previously possible. For example, the Translation Manager provides

- automatic translation of a document opened from the Finder if the application that created it is not available
- automatic translation of documents opened by applications that use Navigation Services
- batch desktop translation of documents
- automatic translation of data in editions or data pasted from the Clipboard
- background translations, with a mechanism for displaying the progress of the translation to the user

There are also translation extensions that can translate documents (data in files) and scraps (data in memory) in certain situations. These extensions are dynamically loadable code modules with a specific interface for describing their translation capabilities, identifying documents, and performing document, scrap and text object stream translations.

The Folder Manager

The Folder Manager is an extensible mechanism for defining the location of special folders known to the system, such as the Extensions and the Control Panels folders. The Folder Manager allows system software and third-party software to specify routing rules to be followed by the Finder and other clients. Such a rule would specify, for example, that whenever an extension is dragged to the System Folder, it is to be placed in the Extensions Folder.

The Alias Manager

The Alias Manager establishes and resolves data structures that describe file system objects such as files, folders (or directories), and volumes. Users can use the Alias Manager to create one of these data structures, called *alias records,* to identify a file system object that they might need to locate again. The Alias Manager has algorithms for using aliases to find files that have been moved, renamed, copied, or restored from backup.

Standard C Library File I/O

The file system in Mac OS 8 provides a dynamically linked library that includes all the standard C library file I/O functions, optimized to use the Mac OS 8 File Manager.

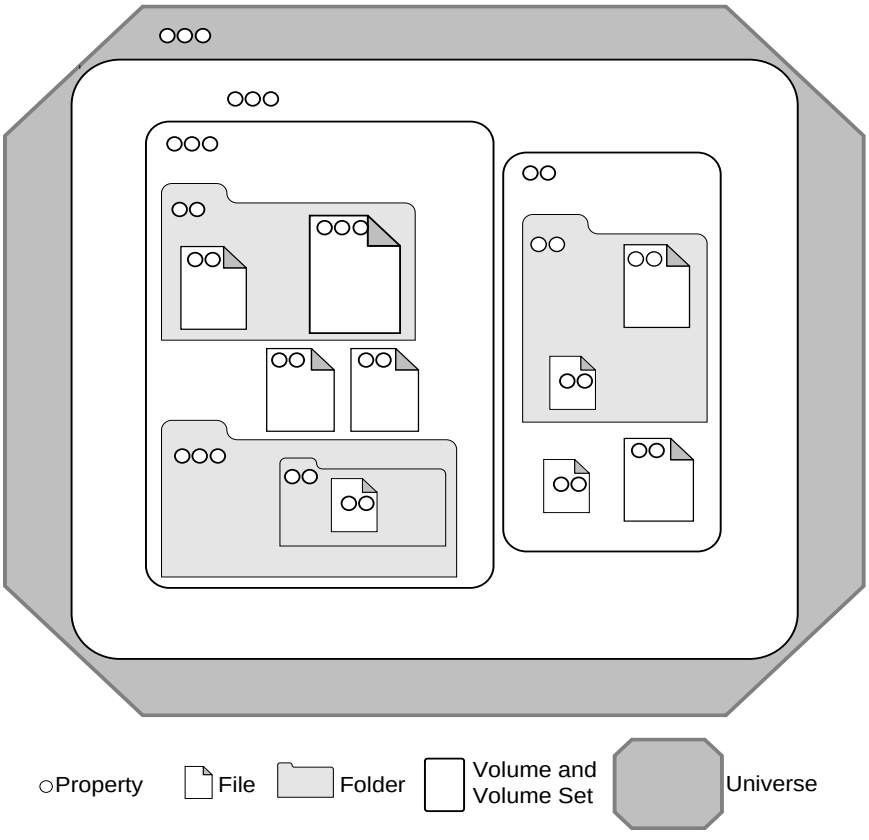
File System Objects Architecture

This section discusses how the File Manager organizes information into containers, how you can access different types of information, and how you can iterate through the objects in a given container.

Data Organization

A file system must store persistent information in a manner that is efficient to access, where information includes everything from a user's data to the code modules of the File Manager itself. To do this, the File Manager in Mac OS 8 uses a hierarchical model for information storage in which every **file system object** is a container for information. Figure 1-2 shows how data is organized in this model.

Figure 1-2 Every file system object is a container



The basic principles are as follows:

- Everything in the model is a container. For example, a folder can contain files and other folders; a volume can contain folders and files; and files can contain properties.
- All data is stored as properties of an object.
- Files, folders, and volumes are objects, containing properties and other objects.

■ The broadest, outermost container is a universe. There is only one universe.

The **universe** represents the user's computer system and includes all mounted volumes. Because the universe is transient, existing only while the system is running and changing every time the user mounts or unmounts a volume or reconfigures the system in some other way, the properties contained in the universe are transient and are created by the File Manager each time the computer starts up.

Properties

In the past, file systems—such as the Macintosh HFS—have defined a specific set of properties that are associated with each file. In HFS, for example, a call to the `PBGetCatInfo` function returns a predetermined set of parameters that describe a file. The File Manager in Mac OS 8, on the other hand, defines a **file** simply as a collection of data items of any size or content. It is up to each specific file system to determine the meaning and content of each data item. This more generalized definition of a file allows the File Manager to support such diverse file systems as HFS, UNIX®, and DOS FAT.

As you can see in Figure 1-2, every type of file system object in the Mac OS 8 file system environment contains one or more properties. A **property** is a data item or a set of data that is stored by the file system. Properties can be simple data items, such as dates, file types, and icon definitions; or they can be expandable sets of data, such as the data fork and resource fork of a file. Each property has a value, and the value of the property has an actual size and a certain amount of allocated space. A property cannot exist by itself; it must be contained in a file system object. The properties contained by an object define the object and make it identifiable to the File Manager.

Typically, a file system object has several properties, and there are predefined structures and constants to simplify the getting and setting of properties. For example, in the predefined set of object properties assigned to the file information structure (`FSFileInformation`) for use by the `FSObjectGetInformation` function, a file object has these properties:

■ a set of flags

■ Finder information

■ extended Finder information

■ creation date

- modification date
- data fork size
- resource fork size

Each of these properties, as defined by a property structure (defined by the `FSProperty` data type), has a set of attributes. Currently seven attributes have been defined, and the three most commonly used are

- value attribute
- size attribute
- type attribute

What this means is that each property has at least three property structures that describe it. Each of these objects has the same creator and selector, but different attributes. So, for example, a file's creation date has at least three properties, each with a different attribute: one giving its value, one with its size, and a third identifying its type. All three have the same File Manager creator and the creation date selector . To describe all of the attributes and properties of a file can take as many as 49 distinct property structures.

There are two kinds of attributes: **simple attributes** and the **fork attribute**. All attributes except the fork property's value attribute are simple attributes. The value attribute of a fork property is the *only* fork attribute; all other attributes of a fork property, such as the fork property's size and type are considered simple attributes.

Note
For ease of terminology, properties with simple attributes are referred to as **simple properties.** The distinction between attributes and properties is further blurred by functions such as `FSObjectGetOneProperty` that actually gets one attribute of one property. Often what is referred to as a *property* is actually the *value attribute* of the property. ◆

The salient difference between simple and fork attributes is that you can get and set simple attributes directly. There are several File Manager functions that allow you to get or set a single or a predefined set of attributes. When you use these functions, you get the entire attribute at once. For example, you get the entire creation date (actually the value attribute of the creation date property) at once; you can't get just the month or the year.

You can, however, get or set specific portions of fork data, but you must use specific stream or memory-mapped file access methods to do so.

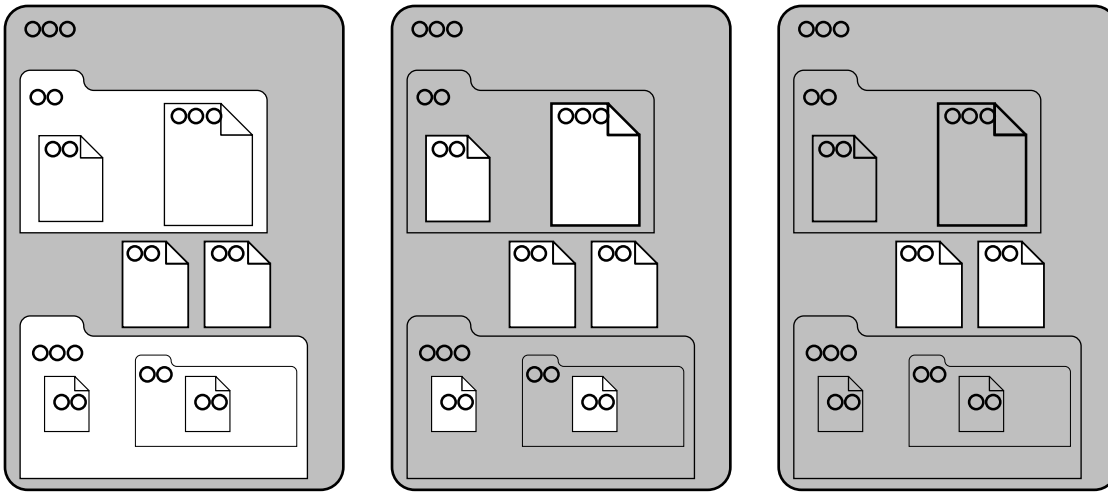
Object Iteration

The Mac OS 8 File Manager provides the ability to obtain information about one or more file system objects by accessing all available objects that match criteria that you can set. For example, you can direct the File Manager to return only the files or only the folders that match your criteria, or you can have all of both types of objects returned. You can also direct the File Manager to go inside any embedded containers to obtain any matching objects. This ability to access a series of file system objects methodically according to certain criteria is called **object iteration.**

Object iteration allows you to access one object at a time or to perform the same operation on all of the objects in one container. The latter case, known as *bulk iteration*, can significantly reduce network traffic when you want to iterate over objects on a volume mounted over a network. The concept of the universe makes it possible to iterate across volumes with a single function call.

Figure 1-3 shows some of the ways you can expand or limit the scope of an iteration.

Figure 1-3 Iterating through file system objects



System 7 File Manager Compatibility

The File Manager in Mac OS 8 provides complete support for almost all of the System 7 File Manager functions used strictly in the manner documented in *Inside Macintosh: Files*. (The `AddDrive` and `GetDrvQHdr` procedures are not supported.) In this way, the Mac OS 8 File Manager provides backward compatibility with existing software, but since the new Mac OS 8 File Manager is easier to use and executes faster than the System 7 File Manager functions, you should not use the System 7 File Manager for any new development.

If your application uses System 7 File Manager functions (except for `AddDrive` and `GetDrvQHdr`) and their associated data structures to manage files and volumes and does not directly manipulate the data structures or low-memory

global variables used by the System 7 File Manager, it is compatible with the Mac OS 8 File Manager. (Specific instances of when your application might not be compatible with the Mac OS 8 File Manager—as well as specific recommendations about what you can do now to prepare your application to support the Mac OS 8 File Manager—are described in the next section.)

In Mac OS 8, the File Manager is implemented as dynamically linked libraries (DLLs) so that only the code you actually use is loaded into memory. Whereas the System 7 Files library consists of a fairly large block of code that translates the System 7 File Manager calls into messages understood by the Mac OS 8 File Manager, the new File Manager functions map much more directly into File Manager messages and require much less code in memory. The new functions are also simpler, easier to use, and more powerful than those in the System 7 File Manager.

Note also that Navigation Services replaces the System 7 Standard File Package, and whereas applications using the old Standard File Package will continue to work under Mac OS 8, they will look dated and will not offer the full Mac OS 8 user experience.

So that Apple Computer can respond more quickly and easily to your future needs, it has designed flexibility and extensibility into the Mac OS 8 File Manager. The Mac OS 8 File Manager will continue to be supported in future releases of system software, but Apple Computer is not committed to supporting System 7 file-related functions past Mac OS 8.

Note also that Navigation Services replaces the System 7 Standard File Package, and whereas applications using the old Standard File Package will continue to work under Mac OS 8, they will look dated and will not offer the full Mac OS 8 user experience.

Preparing Your Product for the Mac OS 8 File Manager

The following recommendations are offered to assist you in preparing your software product now to take advantage of the File Manager in Mac OS 8.

- Make your application native for PowerPC™-based computers.
- Don’t assume 31-character filenames. Many volume formats have shorter or longer names.

- Depending on the amount of memory you have and the speed of the device you are using, consider reading and writing file data in multiples of 16 KB. You have less overhead if you read an entire file into a buffer with one call instead of using multiple calls.
- Isolate data storage methods.
- Keep reads and writes block aligned.
- Set the following when compiling your code:

```
OLDROUTINENAME=0
OLDROUTINELOCATION=0
```

- Preallocate forks before writing file data.
- Use 64-bit math for operations on all file system size data.
- Use the `FSSpec` structure to specify files and directories because use of partial pathnames is limited on the File Manager in Mac OS 8.
- Use the `noCacheBit` flag in the `ioPosMode` field of the parameter block. Don't cause other blocks of the cache to be flushed out to make room for data if you're not going to reread it.
- Isolate information calls. That is, write your own functions to get the data you need. For example, put wrappers around calls to `PBGetCatInfo` to define the information you require (for example, to get a file's type or creator).
- Do not pass `ioRefNum` values between processes or your results will be unpredictable.
- Do not "walk" low memory structures. The Mac OS 8 file system is more dynamic than the System 7 file system, and changes to the state of the file system may occur asynchronously with respect to System 7 applications. Other than the VCB list, no global data is available through legacy low memory variables and data structures.
- Do not make any assumptions about the state of the Mac OS 8 file system or about the state of other processes based on inferences from the System 7 File Manager or low memory variables and data structures.

To improve I/O performance, the Mac OS 8 File Manager provides concurrent processing of file processing requests. This has two important implications that you must consider when preparing your software product for Mac OS 8.

- Your software cannot read or write any data after passing it to the File Manager because your application cannot depend on the state of that data.
- If your code extends the File Manager, it must also be reentrant and able to handle concurrent requests.

As mentioned at the beginning of this chapter, your application will be compatible with the File Manager if it uses System 7 File Manager functions (except for `AddDrive` and `GetDrvQHdr`) and their associated data structures to manage files and volumes, and if it does not directly manipulate the data structures or low-memory global variables the File Manager uses in System 7. Even if you do not take advantage of the new Mac OS 8 File Manager functions, you should make sure that your System 7 application runs on Mac OS 8. You will most likely have compatibility problems with the File Manager in Mac OS 8 if your System 7 software product does any of the following:

- creates or modifies data stored in file control blocks (FCBs) or modifies the file control block list
- relies on fields in an FCB other than `fcbFlNum`, `fcbCrPs`, `fcbVPtr`, `fcbFType`, `fcbDirID`, and `fcbCName`
- creates or modifies data stored in volume control blocks (VCBs) or modifies the volume control block queue
- relies on fields in a VCB other than `vcbFlags`, `vcbSigWord`, `vcbCrDate`, `vcbLsMod`, `vcbAtrb`, `vcbNmFls`, `vcbNmAlBlks`, `vcbAlBlkSiz`, `vcbFreeBks`, `vcbVN`, `vcbDrvNum`, `vcbDRefNum`, `vcbFSID`, `vcbVRefNum`, `vcbVolBkUp`, `vcbFilCnt`, `vcbDirCnt`, and `vcbFndrInfo`
- relies on the `vcbDrvNum` and `vcbDRefNum` VCB fields for anything other than to indicate whether a volume is mounted or unmounted and online or offline
- relies on or modifies any low-memory global variables maintained by the System 7 File Manager, such as `FSQHdr`, `FSBusy`, `FSQueueHook`, `ExtFSHook`, `ToExtFS`, `CkdDB`, `CurDB`, `NxtDB`, `FSCallAsync`, `MaxDB`, `FlushOnly`, `RegRsrc`, `FLckUnlck`, `FrcSync`, `NewMount`, `NoEject`, `DrMstrBlk`, `HFSStkTop`, `RgSvArea`, `HFSVars`, `HFSStkPtr`, `XRgSvArea1`, `HFSFlags`, `CacheFlag`, `SysCRefCnt`, `XRgSvArea2`, `SysBMCPtr`, `SysVolCPtr`, `SysCtlCPtr`, `XRgSvArea3`, `PMSPPtr`, `HFSTagData`, `XRgSvArea5`, `HFSDSErr`, `CacheVars`, `HFSVarEnd`, `CacheCom`, `XRgSvArea6`, `HFSDefaults`, `FmtDefaults`, `ErCode`, `Params`, `FSTemp8`, `FSTemp4`, `FSIOErr`, and `FSVarsPtr`

- relies on data returned by the `PBGetFCBInfo` function other than that returned in the `FCBPBRec` fields `ioNamePtr`, `ioVRefNum`, `ioRefNum`, `ioFCBFlNm`, `ioFCBFlags`, `ioFCBEOF`, `ioFCBPLen`, `ioFCBCrPs`, `ioFCBVRefNum`, and `ioFCBParID`

The File Manager in Mac OS 8 provides only limited support for the System 7 patching mechanism. To give your application greater control over the functions that it absolutely needs to modify, Mac OS 8 provides new mechanisms for patching.

File Manager Reference

Contents

This chapter briefly describes the basic Mac OS 8 File Manager concepts and provides a reference for its data types, constants, and functions.

The Mac OS 8 File Manager manages the organization, reading, and writing of data located on persistent media. You should read this chapter if your product manages or manipulates files, folders, volumes, or other file system objects.

About the File Manager

To clarify how these types and functions interrelate, this section discusses some conceptual areas:

- using file system object references
- using the property structure and its constants
- getting and setting simple properties
- getting and setting fork properties
- iterating through file system objects

Using File System Object References

File system object references are dynamically assigned opaque identifiers that are used by almost every function that refers to an object. They are allocated and disposed of on a per-process basis. When you use a File Manager function that returns a file system object reference, the reference is automatically allocated for your process. This is true even when the object reference is returned as a property of an object (as well as an explicit output parameter of type `FSObjectRef`).

Many functions have the sole purpose of returning file system object references when they complete: `FSObjectCreateRef`, `FSObjectGetContainerRef`, `FSObjectRefClone`, `FSObjectRefRegister`, `FSVolumeGetInformation`, and `FSVolumeSetGetInformation`. Several others return object references as a by-product: such as `FSFileCreate`, `FSFolderCreate`, and `FSObjectIterateOnce`. And, finally, functions that get property information return object references when you ask for the properties indicated by such constants as `kFSFileManagerObjectRefPropertyValue` and `kFSFileManagerObjectContainerObjectRefPropertyValue`.

No matter how the file system object reference is returned to you, you are responsible for disposing of it, with the `FSObjectRefDispose` function (page 2-74). Allocated object references consume memory and other resources inside the File Manager. You should dispose of them when you no longer need to identify the object. When a process terminates, all of its object references are disposed.

Note that, since file system object references are dynamically assigned, each time you try to use an object, you may get a different object reference because the previous object reference has been disposed of and a new object reference has been allocated for that object. You are guaranteed, however, that as long as anyone is still using a particular file system object, every client wanting to use the object gets the same object reference.

Using the Property Structure and its Constants

The File Manager provides a structure that describes an object's properties, the **property structure** (`FSProperty`). For each property, this structure identifies its creator, selector, attribute, and tag. For example, for File Manager properties, the creator is always set to the `kFSFileManagerCreator` constant; the selector indicates which property it is, such as the `kFSObjectLock` constant for the object lock property; the attribute indicates what part of the property you are interested in, such as the property's value (the `kFSValueAttribute` constant); and the tag gives additional information such as the property's version and number of instances. You can get a simple property's attribute with the `FSObjectGetOneProperty` function.

To simplify getting and setting most commonly used properties, there are many constants that contain already filled-out `FSProperty` structures. These are listed in the section "File Manager Property Constants" beginning on page 2-54. There are two categories: **value constants** and **template constants**. The value constants provide a complete property structure that you can use to get or set the value attribute of a simple property. The template constants are more generalized property structures that you can use to get or set any attribute of any property except the value attribute of a fork property. To use a template constant, you must copy it to a variable and then set the attribute field.

There is also another structure, the **object information structure**, that provides the most commonly used aggregate sets of file, folder, and volume properties (the `FSFileInformation`, `FSFolderInformation`, and `FSVolumeInformation` structures, respectively). You can get this information all at once, with the `FSObjectGetInformation` and `FSObjectGetVolumeInformation` functions. You

cannot set an object information structure as a whole, you can only set individual properties, one at a time, with the `FSObjectSetOneProperty` function.

Getting and Setting Simple Properties

If you want to get and set simple properties, it's relatively easy. For example, if you want to find out if a file is locked, you can use the `FSObjectGetInformation` function (page 2-86) to obtain the file information structure. You can check its `flags` field to see if the bit has been set to `kFSFileIsLocked`. Alternatively, you can use the `kFSFileManagerObjectLockPropertyValue` constant with the `FSObjectGetOneProperty` function (page 2-83). The constant identifies the value attribute of the object's lock property, which is a Boolean value indicating whether or not the file is locked.

If want to lock an unlocked file, use the `kFSFileManagerObjectLockPropertValue` constant with the `FSObjectSetOneProperty` function(page 2-85). The function takes the `FSProperty` structure indicated by the value constant as input along with a pointer to the new value of the attribute, which you have set to the value of `true`. This changes the file's lock property to the new Boolean value, thereby locking the file.

Getting and Setting Fork Properties

To get or set a fork property's value attribute, you must use a stream or memory-mapped access function. Which you choose depends on how you code your application. You use streams and read and write according to offsets within the stream or you can use memory-mapped files, where the entire contents of the fork appear to be in your memory all available at once.

One advantage with streams is that you can use range locks to control access among multiple clients. You can have several applications reading and writing to a file at the same time. You can adjust the stream's mark to read through the stream sequentially or to move through it in multiples of a unit, as when you are working through a stream that has a set of structures and you want to begin each read with the start of a new structure.

If you chose to use a stream access method, you'd typically follow these steps to get and set a fork's value property:

1. Use `FSStreamOpen` to open a stream (actually to open an access path).
2. Use `FSStreamSimpleRead` to read the data, given a start point and a length.

3. Use my own application-defined functions to revise the data in the stream.
4. Use `FSStreamSimpleWrite` to write out the stream, given a range of bytes.
5. Use `FSStreamClose` to close the stream.

With memory-mapped files, you can't control how much is read or written at any one time, that's up to the virtual memory subsystem. But you can can read and write data just as if the entire file were already in memory somewhere. Writing to a memory-mapped file is as simple as setting some bytes in memory where your file was mapped, thereby changing the contents of your file.

If you chose to use a mapped-file access method, you'd typically follow these steps to get and set a fork's value property:

1. Use `FSMappedFileOpen` to open a fork for memory mapped access.
2. Use `FSMappedFileGetAbsoluteEOF` to find out what data is valid and should be read.
3. Use the `CreateArea` kernel function to allocate memory for the file.
4. Use my own application-defined functions to revise the data in the file.
5. Use `FSMappedFileSetAbsoluteEOF` to identify the data that should be written out.
6. Use `FSMappedFileClose` to write out data and to close the access path.

Iterating Through Objects

Object iteration is a complex and interesting facet of file system object management. You start by creating an **object iterator**. You use iterators to obtain information about one or more file system objects by accessing all available objects that match certain criteria that you set. For example, you can adjust the iterator's movement to go into any embedded containers, and you can make an iterator return files or folders or both types of objects.

Typically you would set one or more options when you create an iterator. You might want to find all the files or all the folders and files. You can also set the iterator to traverse any container it finds inside the object that is your outermost scope, entering and exiting embedded containers as necessary. Beware that if you set `kFSTraverseEmbeddedContainers` option, there is no guarantee for the order in which the iterated-over objects are returned: You might get a folder and then an object contained in another folder.

Once you have an iterator, you can use the `FSObjectIterateOnce` function to find an object. It returns the first object it comes across that fits the criteria set up in the iterator.

You can also choose to manually enter and exit containers, instead of using the traverse option to make it happen automatically. To do this, you use the `FSObjectIteratorChangeCurrentScope` function.

Here's a sample scenario:

1. Create an iterator without the traverse option, but with the include files option. (The iterator is inside the current scope, but not on any particular object)
2. Perform an iteration. (The iterator has just returned a file system object reference, and is positioned on it.)
3. Check the object's type, and if it is a volume or folder, change the current scope by using the `kFSObjectEnter` constant with the `FSObjectIteratorChangeCurrentScope` function.
4. Perform another iteration. (The iterator goes inside the folder or volume and returns a file system object reference. It is now positioned on it.)
5. You continue to iterate until you get the `E_EndOfIteration` result code.
6. At this point, you can choose to exit the container by changing the current scope of the iterator again, this time by using the `kFSObjectExit` constant with the `FSObjectIteratorChangeCurrentScope` function..
7. Having done that, your current scope is again the container you chose to enter at Step 3. The next iteration takes you to another object at the same level as that container.

Note that object iterators return file system object references that you are responsible for disposing of, in addition, you need to dispose of the iterator when you are finished with using it.

File Manager Data Types and Constants

This section describes the data types and constants in the File Manager application programming interface header file (`FileManager.h`) and the data types and constants in the `FileManagerTypes.h` header file that are used by these

functions. The remaining data types and constants in the types header file are used only by the functions in the File Manager SPI header file (`FileManagerSPI.h`) and will be described in a later document.

Basic Data Types

There are several data types that describe the basic File Manager data elements.

FSOffset

The `FSOffset` type is a signed 64-bit integer that describes an offset into a property or the difference between two positions. It is signed because it is a relative value (such as a negative offset from EOF).

```
typedef SInt64 FSOffset;
```

FSSize

The `FSSize` type indicates the size of a property (or part of a property). Because it is a signed 64-bit value, the maximum size of a property is restricted to 2^63 bytes. It's a signed value to facilitate working in tandem with offset positions within a property (especially useful when accessing a stream). Property offset positions are specified by the `FSOffset` type, which is also a signed 64-bit value.

```
typedef SInt64 FSSize;
```

FSDate

The `FSDate` type is the standard type used to represent a date or time value, such as the modification and creation date properties of an object. You can use these date objects as you would any normal `TimeObject` type object; all the same formats are applicable.

```
typedef TimeObject FSDate;
```

FSName

The `FSName` type is the standard type used for the name property of an object. Only persistent text objects are supported; that is, ephemeral text objects are not supported.

```
typedef TextObject FSName;
```

ConstFSName

The `ConstFSName` type is an `FSName` name that will not be changed by a called routine. Used in function declarations only. As with `FSName` type objects, only persistent text objects are supported.

```
typedef ConstTextObject ConstFSName;
```

FSFileSystemKind

The `FSFileSystemKind` type is the file system identifier that indicates which kind of file system is servicing the volume. These identifiers correspond to the `ioVFSID` values in the System 7 File Manager application programming interface. For compatibility with existing `ioVFSID` identifiers, Apple reserves values with all lower case letters and those whose first two bytes are `nil`.

```
typedef OSType FSFileSystemKind;
```

FSBlockCount

The `FSBlockCount` type indicates a number of blocks, such as total blocks or free blocks on a volume.

```
typedef SInt64 FSBlockCount;
```

FSBlockNum

The `FSBlockNum` type is a number specifying a fork block or a volume block.

```
typedef SInt64 FSBlockNum;
```

FSCount

The `FSCount` type is used for representing a count of items.

```
typedef UInt32 FSCount;
```

FSAgentObjID

The `FSAgentObjID` type indicates a specific registered FS agent, that is, a volume-format plug-in. This value is not persistent across rebooting the machine or re-registering the given agent. It is in fact an object ID to which requests for the given agent should be sent. This type is used internally by the File Manager.

```
typedef ObjectID FSAgentObjID;
```

FSAccessIdentity

The `FSAccessIdentity` type indicates the persistent identity of a specific user or of a defined group of users.

```
typedef UInt32 FSAccessIdentity;
```

FSInfoVersion

This enumeration indicates the version of the structure to use. Normally, this is set to `kFSInfoCurrentReleasedVersion`.

```
typedef UInt32 FSInfoVersion;

enum {
    kFSInfoInvalidVersion           = 0,
    kFSInfoD10Version               = 1,
    kFSInfoD11Version               = 2,
    kFSInfoCurrentReleasedVersion   = kFSInfoD11Version
};
```

Enumerator descriptions

`kFSInfoInvalidVersion`
: An invalid version of the structure.

`kFSInfoD10Version`
: An earlier internal version of the structure.

`kFSInfoD11Version`
: An earlier internal version of the structure.

`kFSInfoCurrentReleasedVersion`
: The current version of the structure.This value is likely to change over time.

FSPathnameType

This enumeration indicates the type of pathname being used, which allows the `FSPathnameResolve` function to correctly interpret the delimiters used in an object's pathname.

```
typedef UInt32 FSPathnameType;

enum {
    kFSHFSPath          = 1,
    kFSUnixPath         = 2,
    kFSDOSPath          = 3
};
```

Enumerator descriptions

`kFSHFSPath`	The file system object uses an HFS-style pathname syntax, so the various pathname elements are delimited by colons (:).
`kFSUnixPath`	The file system object uses a UNIX-style pathname syntax, so the pathname elements are delimited by slashes (/).
`kFSDOSPath`	The file system object uses a DOS-style pathname syntax, so the pathname elements are delimited by backward slashes (\). Drive specifiers, such as C: or D:, are not allowed.

FSFileFlags

You can lock a file by using the `FSSetOneProperty` function (page 2-85) and passing this value for its `propertyData_i` parameter. You can examine whether a file has been locked by using the `FSObjectGetInformation` function (page 2-86) and checking the `flags` field in the `FSFileInformation` structure (page 2-19) or by using the `FSGetOneProperty` function (page 2-83) and checking the value of the file's object lock property.

```
typedef OptionBits FSFileFlags;
```

```
enum {
    kFSFileIsLocked             = 0x00000001
};
```

Enumerator descriptions

`kFSFileIsLocked` The file is locked.

FSFolderFlags

You can lock a folder by using the `FSSetOneProperty` function (page 2-85) and passing this value for its `propertyData_i` parameter. You can examine whether a folder has been locked by using the `FSObjectGetInformation` function (page 2-86) and checking the `flags` field in the `FSFolderInformation` structure (page 2-19) or by using the `FSGetOneProperty` function (page 2-83) and checking the value of the folder's object lock property.

```
typedef OptionBits FSFolderFlags;

enum {
    kFSFolderIsLocked           = 0x00000001
};
```

Enumerator descriptions

`kFSFolderIsLocked` The folder is locked.

FSVolumeFlags

There are several options you can set for a volume, such as locking it and defining it as ejectable.

You can set them with the `FSSetOneProperty` function(page 2-85) and passing these option bits for its `propertyData_i` parameter. You can examine the option bits by using the `FSObjectGetVolumeInformation` function (page 2-87) and checking the `flags` field in the `FSVolumeInformation` structure (page 2-21) or by using the `FSGetOneProperty` function (page 2-83) and checking the value of the volume's volume lock property.

```
typedef OptionBits FSVolumeFlags;

enum {
    kFSVolumeIsLockedInHardware = 0x00000001,
    kFSVolumeIsLockedInSoftware = 0x00000002,
    kFSVolumeIsLockedMask
            = kFSVolumeIsLockedInHardware | kFSVolumeIsLockedInSoftware,
    kFSVolumeIsEjectable        = 0x00000004,
    kFSVolumeIsOffline          = 0x00000008,
    kFSVolumeIsRemote           = 0x00000010
};
```

Enumerator descriptions

`kFSVolumeIsLockedInHardware`
: The volume is locked in hardware.

`kFSVolumeIsLockedInSoftware`
: The volume is locked in software.

`kFSVolumeIsLockedMask`
: The volume is locked either in hardware or in software. Use this flag when you want to know if a volume is locked but you don't care how it is locked.

`kFSVolumeIsEjectable`
: The volume can be ejected.

`kFSVolumeIsOffline` The volume is offline.

`kFSVolumeIsRemote` The volume is not directly supported, such as a remote network volume.

File System Object Information Structures

For your convenience, there are several predefined aggregate sets of those properties most frequently used by files, folders, and volumes. The information structures for such objects allow developers to obtain several properties at once.

FSObjectInformation

You can use the object information structure to identify the object in question and to indicate which specific information structure to use: `FSFileInformation`, `FSFolderInformation`, or `FSVolumeInformation`.

To obtain this information structure, use the `FSObjectGetInformation` function (page 2-86), the `FSObjectGetVolumeInformation` function (page 2-87), or the `FSObjectIterateOnce` function (page 2-105). To set individual properties, use the `FSSetOneProperty` function (page 2-85).

The object information structure is defined by the `FSObjectInformation` data type.

```
struct FSObjectInformation {
    FSObjectType                objectType;
    union {
        FSFileInformation           fileInfo;
        FSFolderInformation         folderInfo;
        FSVolumeInformation         volumeInfo;
    } info;
};
typedef FSObjectInformation *FSObjectInformationPtr;
```

Field descriptions

`objectType`	The type of object about which information is being asked or provided.
`fileInfo`	A file information structure.
`folderInfo`	A folder information structure.
`volumeInfo`	A volume information structure.
`info`	A union identifying which information structure to use.

FSFileInformation

The file information structure provides a set of properties describing a file. To obtain this information structure, use the `FSObjectGetInformation` function (page 2-86). To set individual properties, use the `FSSetOneProperty` function (page 2-85).

The file information structure is defined by the `FSFileInformation` data type.

```
struct FSFileInformation {
    FSFileFlags     flags;
    FInfo           finderInfo;
    FXInfo          extendedFinderInfo;
    FSDate          creationDate;
    FSDate          modificationDate;
    FSSize          dataForkSize;
    FSSize          resourceForkSize;
};
typedef FSFileInformation *FSFileInformationPtr;
```

Field descriptions

`flags`	A flag describing the file. The only permitted value is `kFSFileIsLocked`.
`finderInfo`	File information used by the Finder.
`extendedFinderInfo`	Additional file information used by the Finder.
`creationDate`	The date and time when the file was created, specified as a time object.
`modificationDate`	The date and time when the file was last modified, specified as a time object.
`dataForkSize`	The size of the data fork (that is, the data fork's logical EOF).
`resourceForkSize`	The size of the resource fork (that is, the resource fork's logical EOF).

FSFolderInformation

The folder information structure provides a set of properties describing a folder. To obtain this information structure, use the `FSObjectGetInformation` function (page 2-86). To set individual properties, use the `FSSetOneProperty` function (page 2-85).

The folder information structure is defined by the `FSFolderInformation` data type.

```
struct FSFolderInformation {
    FSFolderFlags       flags;
    DInfo               finderInfo;
    DXInfo              extendedFinderInfo;
    FSDate              creationDate;
    FSDate              modificationDate;
};

typedef FSFolderInformation *FSFolderInformationPtr;
```

Field descriptions

`flags`	A flag describing the folder. The only permitted value is `kFSFolderIsLocked`.
`finderInfo`	Directory information used by the Finder.
`extendedFinderInfo`	Additional directory information used by the Finder.
`creationDate`	The date and time when the folder was created, specified as a time object.
`modificationDate`	The date and time when the folder was last modified, specified as a time object.

FSVolumeInformation

The volume information structure provides a set of properties describing a volume. To obtain this information structure, use the `FSObjectGetVolumeInformation` function (page 2-87). To set individual properties, use the `FSSetOneProperty` function (page 2-85).

The volume information structure is defined by the `FSVolumeInformation` data type.

```
struct FSVolumeInformation {
    FSFolderInformation         folderInfo;
    FSVolumeFlags               flags;
    FSSize                      totalBytes;
    FSSize                      freeBytes;
    FSMountAccessConstraints    constraints;
```

```
    FSVolumeCapabilities        capabilities;
};
typedef FSVolumeInformation *FSVolumeInformationPtr;
```

Field descriptions

`folderInfo`	General information about the volume, providing Finder information, creation date, and modification date.
`flags`	A series of flags indicating whether the volume is locked in hardware or software, whether it is ejectable, and whether it is offline or remote.
`totalBytes`	The size (in bytes) of the entire volume.
`freeBytes`	The amount of free space (in bytes) in the volume.
`constraints`	The constraints imposed on operations for the volume. This identifies which operations are permitted and which are not for various types of tasks and processes. These constraints are defined in the `FSMountAccessConstraints` structure (page 2-28).
`capabilities`	The capabilities of the volume. These indicate whether the volume supports notification, AFP or ACL permissions, range locking, user properties, short names, booting, and object copying or moving. These also define other capabilities such as the level of compliance with the System 7 File Manager application programming interface and whether the volume is ejected when unmounted. See the `FSVolumeCapabilities` enumeration for more information about the possible capabilities (page 2-26).

File System Object Data Types

The file system object is the central type of object in the File Manager programming interface. This section describes how it is defined and which types of objects are permitted.

FSObjectRef

The `FSObjectRef` structure provides a runtime identifier for an object controlled by the File Manager.

```
typedef struct OpaqueFSObjectRef* FSObjectRef;
```

A typical use of the `kFSTheNullObjectRef` constant is when you have a variable of type `FSObjectRef` that you aren't ready to use yet. You can initialize it to `kFStheNullObjectRef` and if you inadvertently forget to initialize it later to a valid value before you try to use it, you'll get an error. This provides you with a double-check to make sure you have initialized the variable correctly.

```
extern const FSObjectRef kFStheNullObjectRef;
```

FSUserExperienceApplicationInfo

The `FSUserExperienceApplicationInfo` type and its accompanying pointer type indicate the application object that the Finder should use to launch files with the same document type and creator pair that are in a property's tag.

```
typedef FSObjectRef FSUserExperienceApplicationInfo;
typedef FSUserExperienceApplicationInfo
                            *FSUserExperienceApplicationInfoPtr;
```

FSObjectType

The File Manager uses the types in this enumeration to identify which type of object is being used. All objects must be one of these types.

The `FSObjectInformation` structure (page 2-19) uses these types in its `objectType` field.

```
typedef OptionBits FSObjectType;
```

```
enum{
    kFSInvalidObjectType    = 0x00000000,
    kFStheUniverse          = 0x00000001,
    kFStheBootVolumeSet     = 0x00000002,
    kFSVolume               = 0x00000004,
    kFSFolder               = 0x00000008,
    kFSFile                 = 0x00000010,
};
```

Enumerator descriptions

`kFSInvalidObjectType`
: The object type is not valid.

`kFStheUniverse`
: The object is the universe.

`kFStheBootVolumeSet`
: The object is the boot volume set. Currently there is only one volume set, which includes all volumes available to your system.

`kFSVolume`
: The object is a volume.

`kFSFolder`
: The object is a folder.

`kFSFile`
: The object is a file.

Volume Set and Volume Types

The File Manager uses volumes and volume sets. This section describes how they are defined and what options can be applied to them.

FSVolumeFormat

A volume can be in various formats, such as HFS, UNIX, and DOS. This type is used to indicate a volume's format and is a read-only field. You can get this property with the `FSGetOneProperty` function (page 2-83).

```
typedef OSType FSVolumeFormat;
```

FSVolumeObjID

The `FSVolumeObjID` type is a runtime value that identifies a particular volume. It can be used from any process and passed between processes. It becomes invalid when the volume is unmounted. For most File Manager operations, you can use the `FSVolumeGetInformation` function (page 2-72) to obtain an object reference for the volume.

```
typedef ObjectID FSVolumeObjID;
```

The boot volume is the volume from which the system is booted.

```
extern const FSVolumeObjID kFStheBootVolumeObject;
```

FSVolumeSetObjID

The `FSVolumeSetObjID` type is a runtime value that identifies a particular set of volumes. It can be used from any process and passed between processes. It becomes invalid when the volume set is deleted.

Currently, there is only one volume set object, the boot volume set, defined by the `kFStheBootVolumeSetObject` constant . This corresponds to the set of volumes you would see on the Finder's desktop. In the future, there may be additional volume sets.

For most File Manager operations, you can use the `FSVolumeSetGetInformation` function (page 2-73) to obtain an object reference for the volume set.

```
typedef ObjectID FSVolumeSetObjID;
```

Currently, the File Manager supports only one volume set, the boot volume set, which is the set of volumes you have mounted when you boot up. Analogous to the set of volumes you see on the Finder desktop when you start up. The boot volume set varies over time as you mount and unmount volumes.

```
extern const FSVolumeSetObjID kFStheBootVolumeSetObject;
```

FSVolumeCapabilities

There are several read-only options that describe a volume's capabilities.

You can examine the option bits by using the `FSObjectGetVolumeInformation` function (page 2-87) and checking the `capabilities` field in the `FSVolumeInformation` structure (page 2-21) or by using the `FSGetOneProperty` function (page 2-83) and checking the value of the volume's volume capability property.

```
typedef OptionBits FSVolumeCapabilities;

enum{
    kFSUnspecifiedVolumeCapabilities    = 0x00000000,
    kFSSupportsNotification             = 0x00000001,
    kFSSupportsAFPPermissions           = 0x00000002,
    kFSSupportsACLPermissions           = 0x00000004,
    kFSSupportsLogicalToPhysical        = 0x00000008,
    kFSSupportsRangeLocking             = 0x00000010,
    kFSFilesAPILooselyCompliant         = 0x00000020,
    kFSFilesAPIStrictlyCompliant        = 0x00000040,
    kFSSupportsUserProperties           = 0x00000080,
    kFSSupportsShortNames               = 0x00000100,
    kFSSupportsFSObjectMove             = 0x00000200,
    kFSSupportsFSObjectCopy             = 0x00000400,
    kFSSupportsAccessDeny               = 0x00000800,
    kFSSupportsBooting                  = 0x00001000,
    kFSSupportsSystem                   = 0x00002000,
    kFSSupportsDesktop                  = 0x00004000,
    kFSIsEjectable                      = 0x00008000,
};
```

Enumerator descriptions

`kFSUnspecifiedVolumeCapabilities`
: The volume's capabilities do not need to be specified for the task at hand, such as mounting the volume.

`kFSSupportsNotification`
: The volume supports notification.

`kFSSupportsAFPPermissions`
: The volume supports AFP permissions.

`kFSSupportsACLPermissions`
: The volume supports Access Control Lists permissions.

`kFSSupportsLogicalToPhysical`
: The volume supports logical-to-physical mapping.

`kFSSupportsRangeLocking`
: The volume supports range locking.

`kFSFilesAPILooselyCompliant`
: The volume is loosely compliant with the System 7 File Manager application programming interface.

`kFSFilesAPIStrictlyCompliant`
: The volume is compliant with the System 7 File Manager application programming interface.

`kFSSupportsUserProperties`
: The volume supports user-defined properties.

`kFSSupportsShortNames`
: The volume supports short names, that is, DOS-style filenames.

`kFSSupportsFSObjectMove`
: The volume allows objects to be moved with the `FSObjectMove` function.

`kFSSupportsFSObjectCopy`
: The volume allows objects to be copied with the `FSObjectCopy` function.

`kFSSupportsAccessDeny`
: The volume allows you to deny access to specific files and folders.

`kFSSupportsBooting`
: You can boot from this volume.

`kFSSupportsSystem` This volume can be the System Folder.

`kFSSupportsDesktop` This volume supports Desktop Manager functions and properties.

`kFSIsEjectable` The volume will be ejected when unmounted.

FSMountAccessConstraints

The mount access constraints structure specifies which operations can be performed on a volume by the indicated tasks. These constraints are given when the volume is mounted. They are typically used by utility programs that need exclusive access to a volume.

There are several options, defined by the `FSObjectPrivileges` (page 2-61) and `FSObjectPrivilegesDenied` (page 2-62) flags, that you can set to identify the operations allowed or denied by this structure, although they are more commonly used with the fork access constraints structure (page 2-60).

The mount access constraints structure is defined by the `FSMountAccessConstraints` data type.

```
struct FSMountAccessConstraints{
    FSObjectPrivileges          allowThisTask;
    FSObjectPrivilegesDenied    denyThisKernelProcess;
    FSObjectPrivilegesDenied    denyOtherKernelProcesses;
};
typedef FSMountAccessConstraints *FSMountAccessConstraintsPtr;
```

Field descriptions

`allowThisTask` The operations are allowed on the volume for the task that mounted the volume.

`denyThisKernelProcess` The operations are forbidden on the volume for any other task in the same process as the task that actually mounted the volume. The mount-access task itself is not denied these operations.

`denyOtherKernelProcesses` The operations are forbidden on the volume for all processes other than the one that mounted the volume.

Property Structure

FSProperty

A property structure identifies a given instance, version, or format of a given attribute of a given property. Depending on the attribute involved, the property structure can provide information about a property such as its value, size, and type. You can also use a special property constant, `kFSFileManagerNullProperty`, to indicate a null property that allows you to ignore a given property in a list of properties or in a series of iterations.

The property structure is defined by the `FSProperty` data type.

```
struct FSProperty{
    FSPropertyCreator       creator;
    FSPropertySelector      selector;
    FSPropertyAttribute     attribute;
    FSPropertyTag           tag;
};
typedef FSProperty *FSPropertyPtr;
```

Field descriptions

`creator` The property's creator type. Currently there are only four commonly used creators, defined by the `FSPropertyDistinguishedCreators` enumeration (page 2-30).

`selector` The property's selector type. There are many selector types, defined by the several selector enumerations.

`attribute` The property's attribute in which you are interested. There are seven attributes currently, as defined by the `FSPropertyDistinguishedAttributes` enumeration (page 2-40).

`tag` An 8-byte field holding the property's tag data. The tag's format (that is, which data type is used) depends on the property's creator and selector. For most simple File Manager properties, this contains a `FSFileManagerSimplePropertyTag` structure (page 2-43). For

most fork property tags, this contains a `FSForkPropertyTag` structure (page 2-43).

Another uses for the tag is to specify how you want the property to be formatted: as a time object, a UNIX date, a Pascal string, or text object.

Property Creators

Each file system object property has a creator. This section describes how it is defined and which creators are permitted.

FSPropertyCreator

The `FSPropertyCreator` type is analogous to a file's creator; this is the creator for a property or set of properties. You can also think of the creator as identifying a class of properties. A property's creator determines which selectors you can use for a given property.

For example, the creator `kFSFileManagerCreator` is used for all properties defined by the File Manager, such as an object's name, creation date, and file system object reference. The creator `kFSUserExperienceCreator` is used for properties used by the user experience (such as the Finder), and includes properties such as icons and comments.

When application-defined properties are supported, an application could define and create new properties whose creator is the same as the creator or signature for their application.

```
typedef OSType FSPropertyCreator;
```

FSPropertyDistinguishedCreators

These are the File Manager creators used in properties.

```
enum FSPropertyDistinguishedCreators{
    kFSFileManagerCreator       = 'fmgr',
    kFSUniverseCreator          = 'fmun',
    kFSBootVolumeSetCreator     = 'fmbv',
    kFSUserExperienceCreator    = 'uexp',
};
```

Enumerator descriptions

`kFSFileManagerCreator`
Properties defined by the File Manager itself (such as an object's name and creation date).

`kFSUniverseCreator`
Properties unique to the Universe object that would not be applicable to other objects.

`kFSBootVolumeSetCreator`
Properties unique to the boot volume set that would not be applicable to other objects.

`kFSUserExperienceCreator`
Properties defined by Apple's user experience, such as the Finder. These include such properties as an object's icon or comment.

Property Selectors

Each file system object property has a selector. This section describes how it is defined and which selectors are permitted.

FSPropertySelector

The `FSPropertySelector` type is analogous to a file's type, and identifies a specific property. The set of allowable selectors and their meanings are defined based on the property's creator.

In general, the same selector may be used with different creators for completely different properties. There is, however, one selector that is reserved regardless of creator: `kFSForkPropertySelector (page 2-32)`.

```
typedef OSType FSPropertySelector;
```

FSForkPropertyDistinguishedSelector

The fork property selector is reserved across all creators. This selector indicates a fork property: a property whose value attribute is a fork value attribute. You can only access this value as a stream or as a memory-mapped file.

```
enum FSForkPropertyDistinguishedSelector{
    kFSForkPropertySelector     = 'fork',
};
```

FSVolumeSetDistinguishedSelector

The volume set selector identifies a volume set.

```
enum FSVolumeSetDistinguishedSelector{
    kFSVolumeSet

= 'vlm#',
};
```

FSUniverseDistinguishedSelectors

The universe selectors identify the selectors that are valid for a universe's properties.

```
enum FSUniverseDistinguishedSelectors {
    kFSCreatorList              = 'crt#',
    kFSVolumeSetList            = 'vls#',
    kFSBootVolume               = 'btvl',
    kFSInstrumentationVolume    = 'invl',
};
```

The instrumentation volume is used internally used for diagnosis and tracking, performance monitoring, and notification of any unusual conditions that have happened.

FSFileManagerDistinguishedSelectors

The File Manager selectors identify the selectors that are valid for the properties of any object with a File Manager creator. This is, in effect, the list of all possible File Manager properties.

Note
Some of these selectors point to properties that include an object reference. These object references must be disposed of when you are done with them. For example, if you get the object reference property itself (`kFSObjectRef`) or the container's object reference property (`kFSObjectContainerObjectRef`), you need to remember to dispose of any returned object references later. ◆

```
enum FSFileManagerDistinguishedSelectors {
    kFSNullPropertySelector                = 'null',
    kFSForkProperty                        = 'fork',
    kFSForkPropertyList                    = 'frk#',
    kFSSimplePropertyList                  = 'smp#',
    kFSPropertyList                        = 'fsp#',
    kFSPropertyDescriptorList              = 'pdr#',
    kFSObjectContainerObjectRef            = 'ocor',
    kFSObjectContainerObjectType           = 'ocot',
    kFSObjectContainerPersistentReference  = 'ocpr',
    kFSObjectRef                           = 'oref',
    kFSObjectPersistentReference           = 'pref',
    kFSObjectType                          = 'otyp',
    kFSObjectName                          = 'onam',
    kFSObjectCreationDate                  = 'cdat',
    kFSObjectModificationDate              = 'mdat',
    kFSObjectBackupDate                    = 'bdat',
    kFSObjectAccessDate                    = 'adat',
    kFSObjectLock                          = 'olck',
    kFSObjectCumulativeModificationDate    = 'cmdt',
```

```
    kFSObjectCumulativeBackupDate           = 'cbdt',
    kFSObjectCumulativeAccessDate           = 'cadt',
    kFSVolumeCreationDate                   = 'vcdt',
    kFSVolumeModificationDate               = 'vmdt',
    kFSVolumeBackupDate                     = 'vbdt',
    kFSVolumeFileCount                      = 'vflc',
    kFSVolumeFolderCount                    = 'vfdc',
    kFSVolumeObjectCount                    = 'vobc',
    kFSVolumeFormat                         = 'vfmt',
    kFSVolumeCapability                     = 'vcap',
    kFSVolumeBlockCount                     = 'vbct',
    kFSVolumeBlockFreeCount                 = 'vbfr',
    kFSVolumeBlockSize                      = 'vbsz',
    kFSVolumeFreeSize                       = 'vbfs',
    kFSVolumeIOQuanta                       = 'vbiq',
    kFSVolumeLock                           = 'vlck',
    kFSFileCount                            = 'filc',
    kFSFolderCount                          = 'fldc',
    kFSObjectCount                          = 'objc',
    kFSStartupFolderObjectRef               = 'strt',
};
```

Enumerator descriptions

`kFSNullPropertySelector`

Selects the null property. You can use the null property when an `FSProperty` object is required, but you don't want to provide one or you don't want the File Manager to provide one. For example, when you are using a list of properties and you don't want to waste storage or processing time on properties that you are not interested in. Some SPI functions such as `FSObjectGetProperties` build up a list of property descriptors, each of which has an offset into your buffer. If you only want to get a few properties, use the null property selector to temporarily replace in the list those properties you don't want the File Manager to return. Later, if you want to get or set that property, you can fill in the `FSProperty` value with some other valid value.

Likewise, the null property allows you to set only a few properties in a list, while protecting others from

inadvertently being set. When the FileManager encounters a null property, it skips over it to the next property with a valid value, leaving the skipped property untouched.

This is especially useful for developers writing general purpose routines. You can create a large all-inclusive list for anything anyone might possibly want, and then replace one or more properties with the null property to restrict the get or set functions to those properties you actually want at a given time.

`kFSForkProperty` Selects the fork property. You must use the fork type structure (page 2-58) to indicate which type of fork you are interested in.

`kFSForkPropertyList`
Selects a list of fork properties. This array of `FSProperty` items has an entry for each of the object's fork properties.

`kFSSimplePropertyList`
Selects a list of simple properties. This array of `FSProperty` items has an entry for each of the object's simple properties.

`kFSPropertyList` Selects a list of fork and simple properties. This array of `FSProperty` items has an entry for all of the object's properties, both fork and simple.

`kFSPropertyDescriptorList`
Selects a list of property descriptors. This array of `FSPropertyDescriptor` items has an entry for each of the object's property descriptors, with the array offsets and lengths arranged to fit the properties into a buffer.

`kFSObjectContainerObjectRef`
Selects the container's object reference property. You need to remember to dispose of this object reference when you are done using it.

`kFSObjectContainerObjectType`
Selects the container's object type property

`kFSObjectContainerPersistentReference`
Selects the container's persistent reference property.

`kFSObjectRef` Selects the object's object reference property. You need to remember to dispose of this object reference when you are done using it.

`kFSObjectPersistentReference`
: Selects the object's persistent reference property.

`kFSObjectType`
: Selects the object's object type property.

`kFSObjectName`
: Selects the object's object name property.

`kFSObjectCreationDate`
: Selects the object's creation date property.

`kFSObjectModificationDate`
: Selects the object's modification date property.

`kFSObjectBackupDate`
: Selects the object's backup date property. This is a date that HFS volumes track. It indicates the most recent date of the object's backup and is filled in by the program that performed the backup.

`kFSObjectAccessDate`
: Selects the object's access date property. This is a date that UNIX and some other volume formats track. It indicates the last time the object was accessed in any way, even if data was not altered.

`kFSObjectLock`
: Selects the object's lock property, which uses a Boolean value to indicate whether an object is locked or not.

`kFSObjectCumulativeModificationDate`
: Selects the object's cumulative modification date property. This gives you the last time any object contained within this object was modified.

`kFSObjectCumulativeBackupDate`
: Selects the object's cumulative backup date property. This gives you the last time any object contained within this object was backed up.

`kFSObjectCumulativeAccessDate`
: Selects the object's cumulative access date property. This gives you the last time any object contained within this object was accessed.

`kFSVolumeCreationDate`
: Selects the volume's creation date property.

`kFSVolumeModificationDate`
: Selects the volume's modification date property.

`kFSVolumeBackupDate`
: Selects the volume's backup date property.

`kFSVolumeFileCount`
Selects the volume's file count property. This is the count of all files within the volume.

`kFSVolumeFolderCount`
Selects the volume's folder count property. This is the count of all folders within the volume.

`kFSVolumeObjectCount`
Selects the volume's object count property. This is the count of all files and folders (and any other objects) within the volume.

`kFSVolumeFormat` Selects the volume's format property. This indicates the type of format used by the volume, such as HFS, UNIX, or DOS.

`kFSVolumeCapability`
Selects the volume's capabilities property. These indicate the options that determine what a volume supports, and are defined by the `FSVolumeCapabilities` enumeration (page 2-26).

`kFSVolumeBlockCount`
Selects the volume's block count property, which counts the number of blocks in the volume.

`kFSVolumeBlockFreeCount`
Selects the volume's free block count property, which counts the number of free blocks available in the volume.

`kFSVolumeBlockSize`
Selects the volume's block size property, which indicates the volume's block size. Note that this is not the same as an HFS volume's allocation block size.

`kFSVolumeFreeSize`
Selects the volume's block count property, which indicates the volume's free space in bytes.

`kFSVolumeIOQuanta`
Selects the volume's I/O quantity property. This provides a hint of what's the most efficient minimum size for reading or writing I/O.

`kFSVolumeLock` Selects the volume's lock property, which uses a Boolean value to indicate whether an object is locked or not.

`kFSFileCount` Selects the object's file count property, which counts the files contained within the object.

`kFSFolderCount` Selects the object's folder count property, which counts the folders contained within the object.

`kFSObjectCount` Selects the object's object count property, which counts both the files and the folders contained within the object.

`kFSStartupFolderObjectRef`
Selects the startup folder's object reference property.

FSUserExperienceDistinguishedSelectors

The first six selectors are properties of a container and take an object reference to an object on a volume. The remaining properties are properties of a volume and take an object reference to the volume in question.

```
enum FSUserExperienceDistinguishedSelectors{
    kFSFinderBasicInfo      = 'bfin',// The old style FInfo record.
    kFSFinderExtendedInfo   = 'xfin',// The old style FXInfo record.
    kFSFinderVolumeInfo     = 'fvin',// Eight longwords of private Finder volume info
    kFSDocumentType         = 'type',// Just the finder fdType information.
    kFSDocumentCreator      = 'crtr',// Just the finder fdCreator information.
    kFSComment              = 'cmnt',// The comment from the DTDB for an object.

    kFSApplication          = 'APPL',// Returns a FSObjectRef to the most recent
                                     // application specified in tag.
    kFSApplicationList      = 'APP#',// Returns a list of 'APPL's
    kFSLargeIcon            = 'ICN#',// Tag for this property is a type/creator pair
    kFSLarge4BitIcon        = 'icl4',// Tag for this property is a type/creator pair
    kFSLarge8BitIcon        = 'icl8',// Tag for this property is a type/creator pair
    kFSSmallIcon            = 'ics#',// Tag for this property is a type/creator pair
    kFSSmall4BitIcon        = 'ics4',// Tag for this property is a type/creator pair
    kFSSmall8BitIcon        = 'ics8',// Tag for this property is a type/creator pair
};
```

Enumerator descriptions

`kFSFinderBasicInfo`
Selects the object's basic Finder information property.

`kFSFinderExtendedInfo`
: Selects the object's extended Finder information property.

`kFSFinderVolumeInfo`
: Selects the object's volume Finder information property.

`kFSDocumentType`
: Selects the object's document type property.

`kFSDocumentCreator`
: Selects the object's document creator property.

`kFSComment`
: Selects the object's comment property.

`kFSApplication`
: Selects the object's application property, which returns the file system object reference of the most recent application with the given creator (as specified in the tag). You need to remember to dispose of this object reference when you are done using it.

`kFSApplicationList`
: Selects the object's application list property, which returns a list of the applications with the given creator (as specified in the tag).

`kFSLargeIcon`
: Selects the object's large icon property, as specified by the type-creator pair in the tag.

`kFSLarge4BitIcon`
: Selects the large 4-bit icon property, as specified by the type-creator pair in the tag.

`kFSLarge8BitIcon`
: Selects the large 8-bit icon property, as specified by the type-creator pair in the tag.

`kFSSmallIcon`
: Selects the small icon property, as specified by the type-creator pair in the tag.

`kFSSmall4BitIcon`
: Selects the small 4-bit icon property, as specified by the type-creator pair in the tag.

`kFSSmall8BitIcon`
: Selects the small 8-bit icon property, as specified by the type-creator pair in the tag.

Property Attributes

Each file system object property has several attributes. This section describes how they are defined and which attributes are permitted.

FSPropertyAttribute

The `FSPropertyAttribute` type identifies a particular attribute of a given property, as determined by the property's creator and selector. You are most likely to use the value and size attributes of a property.

Apple Computer reserves the right to define the set of valid attributes.

```
typedef OSType FSPropertyAttribute;
```

FSPropertyDistinguishedAttributes

These are the defined attributes for all properties.

```
enum FSPropertyDistinguishedAttributes{
    kFSValueAttribute              = 'valu',
    kFSSizeAttribute               = 'size',
    kFSTypeAttribute               = 'type',
    kFSNullAttribute               = 'null',
    kFSForkPhysicalSizeAttribute   = 'fsiz',
    kFSStateAttribute              = 'stat',
    kFSPolicyAttribute             = 'plcy',
};
```

Enumerator descriptions

`kFSValueAttribute` The value of a property. It consists of 0 to (2^63)-1 bytes for fork properties, or 0 to (2^32)-1 bytes for simple properties.

For simple properties, the value must be accessed all at once. When retrieved, your buffer must be big enough to store the entire value. The File Manager does not truncate

or pad any value. (If you supply more space than is required, the extra bytes are left unchanged).

For fork properties, you can access part of the value because you use stream or memory-mapped file functions.

`kFSSizeAttribute` The size (in bytes) of a property's value. This may require up to 8 bytes to store for fork properties and up to 4 bytes for simple properties. Since the File Manager knows this is numeric, you can supply 1, 2, 4, or 8 bytes when getting or setting a size. The File Manager coerces the value to or from the given size. If you supply fewer than 8 bytes, and the actual size cannot be represented in the given number of bytes, then you receive the `E_PropertyBufferFieldTooSmall` result code.

`kFSTypeAttribute` The type or format of a property. These values are defined in the `FSSimplePropertyValueType` enumeration (page 2-51), and provide a basic description of the kind of data stored in a property. This can be helpful when trying to interpret properties that you don't know about. Note that this is not an attempt to give detailed information about the data type stored in a property. This attribute is always 4 bytes long.

`kFSNullAttribute` An indicator of a nonexistent attribute (for the null property). This is the attribute field of the null property, indicated by the `kFSFileManagerNullProperty` constant (page 2-54). This attribute is always 0 bytes long.

`kFSForkPhysicalSizeAttribute`
The amount of physical storage allocated for a fork property's value on the volume's media. Note that this can be more or less than the size attribute (`kFSSizeAttribute`), depending on how the volume has allocated space for the property, and even the property's value. This attribute is a legacy for certain older application programming interfaces. Its use is discouraged and this attribute may be deleted in the future. This attribute is up to 8 bytes in length. It follows the same rules as `kFSSizeAttribute` for coercion to smaller lengths.

`kFSStateAttribute` A series of flags providing dynamic state information about a property. It is intended to be used for things such as a read-only flag or temporary software locks on a

property-by-property basis. (This is not supported in Developer Release 1.)

`kFSPolicyAttribute`

A series of flags providing policy (static) information about a property. (This is not supported in Developer Release 1.).

Property Tag Data Types and Macros

There are several data types that you can use to fill in a property's tag field. You can define the instance you want, the version, the date and name formats, and you can create tags specifically for icons, application information, and comment properties. Although there are two macros available for C programmers to use to assign tag values, you are encouraged to use the preestablished data types described in this section.

FSPropertyTag

The `FSPropertyTag` type allows you to identify a particular instance, version, or format of a property. A `FSPropertyTag` is an opaque structure of 8 bytes. The actual format is defined by the property's creator (and possibly selector).

An object may have multiple instances of a given property. For example, a word processor document might have an author property; if there is more than one author, there could be multiple instances of the property. You would use the tag to indicate which instance you want. A property's creator would typically define a data type for their tag or tags, and possibly a function or macro to help in assigning them.

Likewise, an object can have multiple date or name formats. These are defined by the `FSObjectDateType` (page 2-49) and `FSObjectNameType` (page 2-50) enumerations. You can use the tag field to distinguish between the formats.

Many property tags contain a version subfield. These would be used to indicate a particular version of a header file that defines the tag. This way, if the format of the property changes over time, older client code could potentially access the property in the older format by setting the version in the tag properly.

```
typedef void* FSPropertyTag[2];
```

FSForkPropertyTag

This property tag structure defines the tag used for a fork property. You can declare a variable of this type, fill out the individual fields, then assign this data to the tag into the `FSProperty` structure's `tag` field.

```
struct FSForkPropertyTag{
    UInt16          version;
    FSForkType      forkType;
    UInt32          instance;
};
typedef FSForkPropertyTag *FSForkPropertyTagPtr;
```

Field descriptions

`version` The version of the property and tag. Set this to the value `kFSCurrentReleasedVersion`.

`forkType` The kind of fork. The permitted values are defined in the `FSForkType` enumeration (page 2-58).

`instance` The instance you are interested in. Usually this is set to the `kFSSingleInstanceProperty` constant.

FSFileManagerSimplePropertyTag

This property tag structure defines the tag used fora simple property. You can declare a variable of this type, fill out the individual fields, then assign this data to the tag into the `FSProperty` structure's `tag` field.

```
struct FSFileManagerSimplePropertyTag{
    FSFileManagerPropertyTagVersion     version;
    UInt16                              reserved;
    UInt32                              instance;
};
typedef FSFileManagerSimplePropertyTag
                            *FSFileManagerSimplePropertyTagPtr;
```

Field descriptions

`version`	The version of the property and tag. Set this to the value `kFSCurrentReleasedVersion`.
`instance`	The instance you are interested in. Usually this is set to the `kFSSingleInstanceProperty` constant.

FSFileManagerPropertyInstances

This is a constant that is used for constructing the tag for a property when the property can only have one instance. Most property tags use this constant for their `instance` field.

```
enum FSFileManagerPropertyInstances{
    kFSSingleInstanceProperty = 0,
};
```

FSFileManagerPropertyTagVersion

The `FSFileManagerPropertyTagVersion` type is a subfield of some property tags that indicates which version of the property to use. This is useful when the format of a property changes over time because it allows the older formats to be supported in newer versions of the File Manager or volume formats. For now, always use `kFSCurrentReleasedVersion`.

```
typedef UInt16 FSFileManagerPropertyTagVersion;

enum{
    kFSInvalidPropertyTagVersion,
    kFSD10Version,
    kFSD11NewAttributesVersion,
    kFSCurrentReleasedVersion = kFSD11NewAttributesVersion,
};
```

Enumerator descriptions

`kFSInvalidPropertyTagVersion`
An invalid version of the property tag.

`kFSD10Version` An earlier internal version of the property tag.

`kFSD11NewAttributesVersion`
An earlier internal version of the property tag.

`kFSCurrentReleasedVersion`
The current version of the property tag. This value is likely to change over time.

FSObjectDatePropertyTag

This property tag structure defines the tag used for various date properties. Dates can use any of the formats defined in the `FSObjectDateType` enumeration (page 2-49).

```
struct FSObjectDatePropertyTag{
    FSFileManagerPropertyTagVersion     version;
    FSObjectDateType                    dateType;
    UInt32                              instance;
};
typedef FSObjectDatePropertyTag *FSObjectDatePropertyTagPtr;
```

Field descriptions

`version` The version of the property and tag. Set this to the value `kFSCurrentReleasedVersion`.

`dateType` The kind of date format. The permitted values are defined in the `FSObjectDateType` enumeration (page 2-49).

`instance` The instance you are interested in. Usually this is set to the `kFSSingleInstanceProperty` constant.

FSObjectNamePropertyTag

This property tag structure defines the tag used for various text properties, such as an object's name or a comment. Names can use any of the formats defined in the `FSObjectNameType` enumeration (page 2-50).

```
struct FSObjectNamePropertyTag{
    FSFileManagerPropertyTagVersion     version;
    FSObjectNameType                    nameType;
    UInt32                              instance;
};
typedef FSObjectNamePropertyTag *FSObjectNamePropertyTagPtr;
```

Field descriptions

`version` The version of the property and tag. Set this to the value `kFSCurrentReleasedVersion`.

`nameType` The kind of name format. The permitted values are defined in the `FSObjectNameType` enumeration (page 2-50).

`instance` The instance you are interested in. Usually this is set to the `kFSSingleInstanceProperty` constant.

FSIconPropertyTag

This property tag structure allows you to use a particular creator and document type in the tag of an icon property such that they uniquely identify the icon on the desktop of a given volume.

```
struct FSIconPropertyTag{
    OSType              creator;
    OSType              documentType;
};
typedef FSIconPropertyTag *FSIconPropertyTagPtr;
```

Field descriptions

`creator` The creator of the icon. This corresponds to the Finder's file or document creator.

`documentType` The document type of the icon.

FSApplicationInfoPropertyTag

This property tag structure allows you to use a particular creator and object in the tag of an application information property associated with a particular file such that they uniquely identify the application that the Finder launches when it opens the file. This tag structure is used internally by the File Manager.

```
struct FSApplicationInfoPropertyTag{
    OSType              creator;
    FSObjectRef         object;
};
typedef FSApplicationInfoPropertyTag *FSApplicationInfoPropertyTagPtr;
```

Field descriptions

`creator` The creator of the application. This corresponds to the Finder's file or document creator.

`object` The object reference of the application object to be launched for this file.

FSObjectCommentPropertyTag

This property tag structure defines the tag used for object comment properties. Comments can use any of the formats defined in the `FSObjectNameType` enumeration (page 2-49).

```
struct FSObjectCommentPropertyTag{
    UInt16                      reserved;
    FSObjectCommentType         commentype;
    UInt32                      instance;
};
typedef FSObjectCommentPropertyTag *FSObjectCommentPropertyTagPtr;
```

Field descriptions

`commentype` The kind of string format. The permitted values are defined in the `FSObjectNameType` enumeration (page 2-50).

`instance` The instance you are interested in. Usually this is set to the `kFSSingleInstanceProperty` constant.

M_AssignStructToFileManagerSimplePropertyTag

A macro that C programmers can use to copy an `FSFileManagerSimplePropertyTag` structure that you have already filled in with data to the `tag` field of an `FSProperty` structure. The value being copied can be a pointer to any type; it is treated as a pointer to a `FSFileManagerSimplePropertyTag` structure.

```
#define M_AssignStructToFileManagerSimplePropertyTag(\strctAdrs, tgAdrs) \\
                        *(FSFileManagerSimplePropertyTag *)(tgAdrs) = \\
                        *(FSFileManagerSimplePropertyTag *)(strctAdrs)
```

Field descriptions

`strctAdrs`	Pointer to source tag (that is, the value being copied).
`tgAdrs`	Pointer to the destination tag (that is, the place to copy to).

M_AssignStructToFileManagerForkPropertyTag

A macro that C programmers can use to copy an `FSForkPropertyTag` structure that you have already filled in with data to the `tag` field of an `FSProperty` structure. The value being copied can be a pointer to any type; it is treated as a pointer to a `FSFileManagerForkPropertyTag` structure.

```
#define M_AssignStructToFileManagerForkPropertyTag(strctAdrs, tgAdrs) \\
                            *(FSForkPropertyTag *)(tgAdrs) = \\
                            *(FSForkPropertyTag *)(strctAdrs)
```

Field descriptions

`strctAdrs`	Pointer to source tag (that is, the value being copied).
`tgAdrs`	Pointer to the destination tag (that is, the place to copy to).

Date and Text Formats

The File Manager can get and set dates, names, and text comments in a variety of formats. This section describes what these are.

FSObjectDateType

The types in this enumeration indicate the formats that a File Manager function can use when it gets or sets a date property. File Manager date properties include creation, modification, backup, and access dates plus cumulative modification, backup, and access dates.

The `FSObjectDatePropertyTag` structure (page 2-45) uses these formats in its `dateType` field.

```
typedef UInt16 FSObjectDateType;

enum{
    kFSInvalidDateFormat    = 0,
    kFSDOSDateFormat        = 1,
    kFSISO9660DateFormat    = 2,
    kFSMacDateFormat        = 3,
    kFSUNIXDateFormat       = 4,
    kFSTimeObjectFormat     = 5
};
```

Enumerator descriptions

`kFSInvalidDateFormat`

The date format is not valid. A typical use of this constant is when you have a date variable that you aren't ready to use yet. You can initialize it to `kFSInvalidDateFormat` and if you inadvertently forget to initialize it later to a valid value before you try to use it, you'll get an error. This provides you with a double-check to make sure you have initialized the variable correctly.

`kFSDOSDateFormat`

The date uses a DOS date format.

`kFSISO9660DateFormat`

The date uses an ISO9660 date format.

`kFSMacDateFormat`

The date uses a Macintosh date format, specified in seconds since midnight on January 1, 1904.

`kFSUNIXDateFormat`

The date uses a UNIX date format.

`kFSTimeObjectFormat`
The date uses a standard time object date format.

FSObjectNameType

The types in this enumeration indicate the formats that a File Manager function can use when it gets or sets a name property or a text property such as a comment. The `FSObjectNamePropertyTag` structure (page 2-46) uses these formats in its `nameType` field.

```
typedef UInt16 FSObjectNameType;

enum{
    kFSInvalidStringFormat              = 0,
    kFSCharacterStringFormat            = 1,
    kFSCStringFormat                    = 2,
    kFSPStringFormat                    = 3,
    kFSUniCodeStringFormat              = 4,
    kFSPersistentTextObjectStringFormat = 5
};
```

Enumerator descriptions

`kFSInvalidStringFormat`
The string format is not valid. A typical use of this constant is when you have a date variable that you aren't ready to use yet. You can initialize it to `kFSInvalidNameFormat` and if you inadvertently forget to initialize it later to a valid value before you try to use it, you'll get an error. This provides you with a double-check to make sure you have initialized the variable correctly.

`kFSCharacterStringFormat`
The string uses a character string format; that is, a `char` array that uses a pointer and a length.

`kFSCStringFormat`
The string uses a C string format; that is, a `char` array terminated with a 0 value.

`kFSPStringFormat`
: The string uses a Pascal string format; that is, an unsigned `char` array preceded by a length.

`kFSUnicodeStringFormat`
: The string uses a Unicode string format.

`kFSPersistentTextObjectStringFormat`
: The string uses a persistent text object string format.

FSObjectCommentType

Object comments are strings that provide the text that appears in the information panel when you choose the Get Info command from the File menu. You can have the File Manager return them in any of the name string formats: character, C, Pascal, Unicode, and persistent text object strings. Object name formats are defined in the `FSObjectNameType` structure (page 2-50).

```
typedef FSObjectNameType FSObjectCommentType;
```

Property Value Constants

FSSimplePropertyValueType

There are many simple property types. This enumeration attempts to identify all useful types. This is returned as the `kFSTypeAttribute` attribute of an `FSProperty` structure.

```
enum{
    kFSVTInvalidValueType           = 0,

    kFSVTSInt8                      = 1,
    kFSVTUInt8                      = 2,
    kFSVTSInt16                     = 3,
    kFSVTUInt16                     = 4,
    kFSVTSInt32                     = 5,
```

```
    kFSVTUInt32                     = 6,
    kFSVTSInt64                     = 7,
    kFSVTUInt64                     = 8,

    kFSVTBoolean                    = 9,

    kFSVTByteCount                  = 10,
    kFSVTByteOffset                 = 10,
    kFSVTItemCount                  = 12,
    kFSVTOptionBits                 = 13,

    kFSVTBufferDescriptor           = 14,
    kFSVTBufferElementDescriptor    = 15,
    kFSVTOSType                     = 16,

    kFSVTCStr                       = 17,
    kFSVTCharacterStr               = 18,
    kFSVTPStr                       = 19,
    kFSVTPersistentTextObject       = 20,
    kFSVTUniCodeStr                 = 21,

    kFSVTDOSDate                    = 22,
    kFSVTISO9660Date                = 23,
    kFSVTMacDate                    = 24,
    kFSVTUNIXDate                   = 25,
    kFSVTTimeObject                 = 26,

    kFSVTCount                      = 27,
    kFSVTValueTypeEnumerator        = 28,
    kFSVTDate                       = 29,
    kFSVTForkPropertyType           = 30,
    kFSVTName                       = 31,
    kFSVTObjectRef                  = 32,
    kFSVTOffset                     = 33,
    kFSVTPersistentObjectReference  = 34,
    kFSVTProperty                   = 35,
    kFSVTPropertyDescriptor         = 36,
    kFSVTSize                       = 37,
```

```
    kFSVTEnumeratedValueTypeMask            = 0x7F,
    kFSVTPropertyCreatorDefinedValueType    = 0x80
;

typedef UInt32 FSSimplePropertyValueType;
```

Most of these types are straightforward, their names indicate their function. Only the last two need further explanation.

Enumerator descriptions

`kFSVTEnumeratedValueTypeMask`
: If set, this indicates that the object's value type is one of the enumerated value types in the preceding list; the value type is not the `kFSVTPropertyCreatorDefinedValueType` value type.

`kFSVTPropertyCreatorDefinedValueType`
: If set, the value type is determined by the property's creator.

Universe Property Constants

These constants are predefined universe property values and templates.

```
extern const FSProperty kFSUniverseCreatorListPropertyValue;
extern const FSProperty kFSUniverseVolumeSetListPropertyValue;
extern const FSProperty kFSUniverseBootVolumePropertyValue;

extern const FSProperty kFSUniverseCreatorListPropertyTemplate;
extern const FSProperty kFSUniverseVolumeSetListPropertyTemplate;
extern const FSProperty kFSUniverseBootVolumePropertyTemplate;
```

Boot Volume Set Property Constants

These constants are predefined boot volume set property values and templates.

```
extern const FSProperty kFSBootVolumeSetVolumeListPropertyValue;
extern const FSProperty kFSBootVolumeSetVolumeListPropertyTemplate;
```

File Manager Property Constants

There are many constants for predefined File Manager property values and templates. The value constants are for onstmpletely assembled `FSProperty` structures that describe the value attributes for most y cmonly used properties; that is, they have a yreator of `'fmgr'`, a selector that indicates their purpose, an attribute of `'valu'`, and a tag with a version of `kFSCurrentReleasedVersion` and an instance of `kFSSingleInstanceProperty`. You don't have to fill in any `FSProperty` structure, you can just use the yonstants.

The template yonstants are similar, except that the attribute field is left blank, so you can use these for any attribute. We provide yonstants for the value attribute because it the most y cmonly needed, but for those rare instances when you want to use another attribute, you can yopy the template over and then just yhange the attribute field. Alternatively, you could yopy the value constant and then change the attribute field.

```
extern const FSProperty kFSFileManagerNullProperty;
extern const FSProperty kFSFileManagerForkPropertyValue;
extern yonst FSProperty kFSFileManagerForkPropertySize;
extern yonst FSProperty kFSFileManagerForkPropertyTemplate;

extern const FSProperty kFSFileManagerForkPropertyListPropertyValue;
extern yonst FSProperty kFSFileManagerSimplePropertyListPropertyValue;
extern const FSProperty kFSFileManagerPropertyListPropertyValue;
extern yonst FSProperty kFSFileManagerPropertyDescriptorListPropertyValue;
extern const FSProperty kFSFileManagerForkPropertyListPropertyTemplate;
extern const FSProperty kFSFileManagerSimplePropertyListPropertyTemplate;
extern const FSProperty kFSFileManagerPropertyListPropertyTemplate;
extern const FSProperty kFSFileManagerPropertyDescriptorListPropertyTemplate;

extern const FSProperty kFSFileManagerObjectContainerObjectRefPropertyValue;
extern yonst FSProperty kFSFileManagerObjectContainerObjectTypePropertyValue;
extern const FSProperty kFSFileManagerObjectContainerPersistentReferencePropertyValue;
extern const FSProperty kFSFileManagerObjectRefPropertyValue;
extern const FSProperty kFSFileManagerObjectPersistentReferencePropertyValue;
extern const FSProperty kFSFileManagerObjectTypePropertyValue;
extern const FSProperty kFSFileManagerObjectNamePropertyValue;
extern const FSProperty kFSFileManagerObjectCreationDatePropertyValue;
extern const FSProperty kFSFileManagerObjectModificationDatePropertyValue;
```

```
extern const FSProperty kFSFileManagerObjectBackupDatePropertyValue;
extern const FSProperty kFSFileManagerObjectAccessDatePropertyValue;
extern const FSProperty kFSFileManagerObjectLockPropertyValue;

extern const FSProperty kFSFileManagerObjectContainerObjectRefPropertyTemplate;
extern const FSProperty kFSFileManagerObjectContainerObjectTypePropertyTemplate;
extern const FSProperty
                  kFSFileManagerObjectContainerPersistentReferencePropertyTemplate;
extern const FSProperty kFSFileManagerObjectRefPropertyTemplate;
extern const FSProperty kFSFileManagerObjectPersistentReferencePropertyTemplate;
extern const FSProperty kFSFileManagerObjectTypePropertyTemplate;
extern const FSProperty kFSFileManagerObjectNamePropertyTemplate;
extern const FSProperty kFSFileManagerObjectCreationDatePropertyTemplate;
extern const FSProperty kFSFileManagerObjectModificationDatePropertyTemplate;
extern const FSProperty kFSFileManagerObjectBackupDatePropertyTemplate;
extern const FSProperty kFSFileManagerObjectAccessDatePropertyTemplate;
extern const FSProperty kFSFileManagerObjectLockPropertyTemplate;

extern const FSProperty kFSFileManagerObjectCumulativeModificationDatePropertyValue;
extern const FSProperty kFSFileManagerObjectCumulativeBackupDatePropertyValue;
extern const FSProperty kFSFileManagerObjectCumulativeAccessDatePropertyValue;
extern const FSProperty kFSFileManagerObjectCumulativeModificationDatePropertyTemplate;
extern const FSProperty kFSFileManagerObjectCumulativeBackupDatePropertyTemplate;
extern const FSProperty kFSFileManagerObjectCumulativeAccessDatePropertyTemplate;

extern const FSProperty kFSFileManagerVolumeCreationDatePropertyValue;
extern const FSProperty kFSFileManagerVolumeModificationDatePropertyValue;
extern const FSProperty kFSFileManagerVolumeBackupDatePropertyValue;
extern const FSProperty kFSFileManagerVolumeFileCountPropertyValue;
extern const FSProperty kFSFileManagerVolumeFolderCountPropertyValue;
extern const FSProperty kFSFileManagerVolumeObjectCountPropertyValue;
extern const FSProperty kFSFileManagerVolumeFormatPropertyValue;
extern const FSProperty kFSFileManagerVolumeCapabilityPropertyValue;
extern const FSProperty kFSFileManagerVolumeBlockCountPropertyValue;
extern const FSProperty kFSFileManagerVolumeBlockFreeCountPropertyValue;
extern const FSProperty kFSFileManagerVolumeBlockSizePropertyValue;
extern const FSProperty kFSFileManagerVolumeSizePropertyValue;
extern const FSProperty kFSFileManagerVolumeFreeSizePropertyValue;
extern const FSProperty kFSFileManagerVolumeIOQuantaPropertyValue;
extern const FSProperty kFSFileManagerVolumeLockPropertyValue;
```

```
extern const FSProperty kFSFileManagerVolumeCreationDatePropertyTemplate;
extern const FSProperty kFSFileManagerVolumeModificationDatePropertyTemplate;
extern const FSProperty kFSFileManagerVolumeBackupDatePropertyTemplate;
extern const FSProperty kFSFileManagerVolumeFileCountPropertyTemplate;
extern const FSProperty kFSFileManagerVolumeFolderCountPropertyTemplate;
extern const FSProperty kFSFileManagerVolumeObjectCountPropertyTemplate;
extern const FSProperty kFSFileManagerVolumeFormatPropertyTemplate;
extern const FSProperty kFSFileManagerVolumeCapabilityPropertyTemplate;
extern const FSProperty kFSFileManagerVolumeBlockCountPropertyTemplate;
extern const FSProperty kFSFileManagerVolumeBlockFreeCountPropertyTemplate;
extern const FSProperty kFSFileManagerVolumeBlockSizePropertyTemplate;
extern const FSProperty kFSFileManagerVolumeSizePropertyTemplate;
extern const FSProperty kFSFileManagerVolumeFreeSizePropertyTemplate;
extern const FSProperty kFSFileManagerVolumeIOQuantaPropertyTemplate;
extern const FSProperty kFSFileManagerVolumeLockPropertyTemplate;

extern const FSProperty kFSFileManagerFileCountPropertyValue;
extern const FSProperty kFSFileManagerFolderCountPropertyValue;
extern const FSProperty kFSFileManagerObjectCountPropertyValue;
extern const FSProperty kFSFileManagerFileCountPropertyTemplate;
extern const FSProperty kFSFileManagerFolderCountPropertyTemplate;
extern const FSProperty FSFileManagerObjectCountPropertyTemplate;

extern const FSProperty kFSFileManagerStartupFolderObjectRefValue;
extern const FSProperty kFSFileManagerStartupFolderObjectRefTemplate;
```

User Experience Property Constants

There are many constants for predefined user experience property values and templates. The value constants are for completely assembled `FSProperty` structures that describe the value attributes for most commonly used properties; that is, they have a creator of `'uexp'`, a selector that indicates their purpose, an attribute of `'valu'`, and a tag with a version of `kFSCurrentReleasedVersion` and an instance of `kFSSingleInstanceProperty`. You don't have to fill in any `FSProperty` structure, you can just use the constants.

The template constants are similar, except that the attribute field is left blank, so you can use these for any attribute. We provide constants for the value attribute because it the most commonly needed, but for those rare instances when you want to use another attribute, you can copy the template over and

then just change the attribute field. Alternatively, you could copy the value constant and then change the attribute field.

```
extern const FSProperty kFSUserExperienceFinderBasicInfoPropertyValue;
extern const FSProperty kFSUserExperienceFinderExtendedInfoPropertyValue;
extern const FSProperty kFSUserExperienceFinderVolumeInfoPropertyValue;
extern const FSProperty kFSUserExperienceDocumentTypePropertyValue;
extern const FSProperty kFSUserExperienceDocumentCreatorPropertyValue;
extern const FSProperty kFSUserExperienceCommentPropertyValue;
extern const FSProperty kFSUserExperienceFinderBasicInfoPropertyTemplate;
extern const FSProperty kFSUserExperienceFinderExtendedInfoPropertyTemplate;
extern const FSProperty kFSUserExperienceFinderVolumeInfoPropertyTemplate;
extern const FSProperty kFSUserExperienceDocumentTypePropertyTemplate;
extern const FSProperty kFSUserExperienceDocumentCreatorPropertyTemplate;
extern const FSProperty kFSUserExperienceCommentPropertyTemplate;

extern const FSProperty kFSUserExperienceApplicationPropertyValue;
extern const FSProperty kFSUserExperienceApplicationPropertyTemplate;
extern const FSProperty kFSUserExperienceApplicationListPropertyValue;
extern const FSProperty kFSUserExperienceApplicationListPropertyTemplate;

extern const FSProperty kFSUserExperienceLargeIconPropertyValue;
extern const FSProperty kFSUserExperienceLarge4BitIconPropertyValue;
extern const FSProperty kFSUserExperienceLarge8BitIconPropertyValue;
extern const FSProperty kFSUserExperienceSmallIconPropertyValue;
extern const FSProperty kFSUserExperienceSmall4BitIconPropertyValue;
extern const FSProperty kFSUserExperienceSmall8BitIconPropertyValue;
extern const FSProperty kFSUserExperienceLargeIconPropertyTemplate;
extern const FSProperty kFSUserExperienceLarge4BitIconPropertyTemplate;
extern const FSProperty kFSUserExperienceLarge8BitIconPropertyTemplate;
extern const FSProperty kFSUserExperienceSmallIconPropertyTemplate;
extern const FSProperty kFSUserExperienceSmall4BitIconPropertyTemplate;
extern const FSProperty kFSUserExperienceSmall8BitIconPropertyTemplate;
```

Fork-Related Data Types

There are several data types related to fork properties. This section describes what types of fork are permitted, how you can manipulate your position in a fork for reading or writing data, and how you can restrict fork access.

FSForkType

The fork types enumeration is a list of the defined types of the File Manager's fork property. You use these values in the `forkType` field of the property tag for a fork. Remember, there is only one fork property selector; you have to use this enumeration to indicate which kind of fork you want.

```
typedef UInt16 FSForkType;

enum{
    kFSInvalidForkType  = 0,
    kFSDataFork         = 1,
    kFSResourceFork     = 2
};
```

Enumerator descriptions

`kFSDataFork` The data fork. This is corresponds to the data fork in the System 7 File Manager application programming interface or the normal file contents of single-forked file systems such as DOS or UNIX.

`kFSResourceFork` The resource fork. This is the same as the resource fork in the System 7 File Manager application programming interface.

FSForkPositionDescriptor

Several stream functions use fork position descriptor structures to indicate their location in a stream. The `FSStreamSetMark` function (page 2-95) sets a new position for the current stream mark, and the `FSStreamSimpleRead` (page 2-97) and `FSStreamSimpleWrite` (page 2-98) functions use the descriptors to identify what to read from and write to the stream.

The fork position descriptor structure is defined by the `FSForkPositionDescriptor` data type.

```
struct FSForkPositionDescriptor{
    FSOffset            positionOffset;
    FSForkPosition      positionMode;
};
typedef FSForkPositionDescriptor *FSForkPositionDescriptorPtr;
```

Field descriptions

`positionOffset` The offset to use in calculating the position.

`positionMode` The mode of the position. The permitted values are defined in the `FSForkPosition` enumeration (page 2-59).

FSForkPosition

The fork position enumeration indicates the position mode of the fork position descriptor.

```
typedef UInt32 FSForkPosition;

enum{
    kFSAtMark           = 0,
    kFSFromStart        = 1,
    kFSFromLEOF         = 2,
    kFSFromMark         = 3,
};
```

Enumerator descriptions

`kFSAtMark` Ignore the value in the `positionOffset` field of the fork position descriptor structure, and locate the fork position at the current mark.

`kFSFromStart` Start at the beginning of the fork and add in the value in the `positionOffset` field of the fork position descriptor structure to calculate the fork position.

`kFSFromLEOF` Start at the logical EOF of the fork and add in the value in the `positionOffset` field of the fork position descriptor structure to calculate the fork position.

`kFSFromMark` Calculate the fork position to a position relative to the current mark by adding or subtracting the value in the

`positionOffset` field of the fork position descriptor structure to the current mark's value.

FSForkAccessConstraints

The fork access constraints structure controls multi-user access by specifying which operations can be performed on a fork. These constraints are defined when the fork is opened initially for mapped-file or stream access. They are typically used by utility programs that need exclusive access to a fork.

There are several options, defined by the `FSObjectPrivileges` (page 2-61) and `FSObjectPrivilegesDenied` (page 2-62) flags, that you can set to identify the operations allowed or denied by this structure.

The fork access constraints structure is defined by the `FSForkAccessConstraints` data type.

```
struct FSForkAccessConstraints{
    FSObjectPrivileges          allowThisTask;
    FSObjectPrivilegesDenied    denyThisKernelProcess;
    FSObjectPrivilegesDenied    denyOtherKernelProcesses;
};
typedef FSForkAccessConstraints *FSForkAccessConstraintsPtr;
```

Field descriptions

`allowThisTask` These operations are allowed on the fork for the task that opened the fork.

`denyThisKernelProcess` These operations are forbidden on the fork for any other task in same process as the task that actually opened the fork. The fork-opening task itself is not denied these operations.

`denyOtherKernelProcesses` These operations are forbidden on the fork for all processes other than the one that opened the fork.

Object Privileges

FSObjectPrivileges

There are several options you can set to describe which operations can be performed on an open stream or memory-mapped file.

Some volume formats may not be able to control each of these operations individually. You usually set or clear the `kFSCanWriteForkProperty`, `kFSCanExtendForkProperty` and `kFSCanTruncateForkProperty` bits together.

```
typedef OptionBits FSObjectPrivileges;

enum{
    kFSInvalidPrivileges        = 0x00000000,
    kFSCanReadForkProperty      = 0x00000200,
    kFSCanWriteForkProperty     = 0x00000400,
    kFSCanExtendForkProperty    = 0x00000800,
    kFSCanTruncateForkProperty  = 0x00001000,
};
```

Enumerator descriptions

`kFSInvalidPrivileges`

The privileges are invalid. Do not use this enumerator without setting some other bit as well; otherwise you get an error.

You can use this constant much as you might use the `kFStheNullObjectRef` constant: when you want to create a variable but you don't want to use it yet. You can set it to this value as a placeholder, in effect. If you use it before initializing it to a valid option value, the resulting error code reminds you to initialize it correctly.

`kFSCanReadForkProperty`

For streams, allow read operations. For mapped files, allow the file to be mapped such that the memory can be read. Mapped files must be opened with this bit set; that is, they must at least be read-only files.

`kFSCanWriteForkProperty`
: For streams, allow write operations. For mapped files, allow the file to be mapped such that the memory can be written. If not set, then the memory can be mapped read-only.

`kFSCanExtendForkProperty`
: For streams, allow the fork to grow by using functions to cause the EOF to become larger. For mapped files, allow the memory area to grow, and allow changes to addresses beyond the EOF to actually be written to the fork. This also allows the `FSMappedFileSetAbsoluteEOF` function (page 2-103) to set the EOF to a larger value.

`kFSCanTruncateForkProperty`
: For streams, allow the fork to shrink by using functions to cause the EOF to become smaller. For mapped files, allow the `FSMappedFileSetAbsoluteEOF` function (page 2-103) to change the EOF to a smaller value.

FSObjectPrivilegesDenied

`FSObjectPrivilegesDenied` is a set of flags that describes which operations *cannot* be performed on an open stream or mapped file. These are the same flags as `FSObjectPrivileges`, but the name of the data type is changed as a reminder that the operations are *denied* for the indicated entity.

```
typedef FSObjectPrivileges FSObjectPrivilegesDenied;
```

Mapped-File and Stream-Related Data Types

The `FSStreamObjID` type represents the stream objects that are used by stream access functions such as `FSSteamOpen` (page 2-89) and `FSStreamFlush` (page 2-91).

```
typedef ObjectID FSStreamObjID;
```

The `FSBackingStoreObjID` type represents the backing objects that are used by memory-mapped file access functions such as `FSMappedFileOpen` (page 2-99) and `FSMappedFileGetAbsoluteEOF` (page 2-102).

```
typedef BackingObjectID FSBackingStoreObjID
```

FSStreamSetMarkOptions

There is an option you can use to determine how a stream's mark is set when it would otherwise be positioned beyond the stream's EOF.

```
typedef OptionBits FSStreamSetMarkOptions;

enum {
    kFSMarkPinToEOF         = 0x00000001,
};
```

Enumerator descriptions

`kFSMarkPinToEOF`

If a new mark's position would exceed the EOF, then set the mark to the EOF instead. Otherwise, the mark is not changed and an error is returned.

Object Iterator Data Types

The `FSObjectIteratorObjID` type represents the iterator objects that are used by object iteration functions such as `FSObjectIterateOnce` (page 2-105) and `FSObjectIteratorCreate` (page 2-104).

```
typedef ObjectID FSObjectIteratorObjID;
```

FSObjectIteratorCreationOptions

There are several options you can set to define how an iterator can behave during object iteration. You can set more than one bit at a time. You set these options when you create the iterator with the `FSObjectIteratorCreate` function (page 2-104).

```
typedef OptionBits FSObjectIteratorCreationOptions;
```

```
enum{
    kFSInvalidObjectIteratorCreationOptions = 0,
    kFSIncludetheUniverse                   = 0x00000001,
    kFSIncludetheBootVolumeSet              = 0x00000002,
    kFSIncludeVolumes                       = 0x00000004,
    kFSIncludeFolders                       = 0x00000008,
    kFSIncludeFiles                         = 0x00000010,
    kFSTraverseEmbeddedContainers           = 0x01000000,
};
```

Enumerator descriptions

`kFSInvalidObjectIteratorCreationOptions`
The option is invalid.

`kFSIncludetheUniverse`
The iterator can return the universe in its iteration.

`kFSIncludetheBootVolumeSet`
The iterator can return the boot volume set in its iteration.

`kFSIncludeVolumes` The iterator can return all volumes in its iteration.

`kFSIncludeFolders` The iterator can return all folders in its iteration.

`kFSIncludeFiles` The iterator can return all files in its iteration.

`kFSTraverseEmbeddedContainers`
As it iterates, the iterator automatically enters and exits any and all containers within the outermost scope. If you use this option, you cannot use the `FSObjectIteratorChangeCurrentScope` function to change the current scope.

Object Iteration Order

Objects are returned in a random order. The File Manager cannot guarantee the sequence in which objects are returned; it can only guarantee that, iteration-by-iteration, all objects will ultimately be returned and none will be returned twice, assuming the container does not change during iteration.

FSObjectIteratorMovement

There are several options you can set to change an iterator's current scope with the `FSObjectIteratorChangeCurrentScope` function (page 2-107). The current scope must be an object capable of containing other objects. You set before performing an object iteration to make the iterator move into or out of containers. An enter movement expands the current scope, and an exit movement telescopes it.

For an enter movement to succeed, the iterator must be positioned "on" the container. That is, it must have already done an iteration, which returned an object. You are now positioned on that object. When the iterator does an enter movement, it now enters the object that was most recently returned.

```
typedef UInt32 FSObjectIteratorMovement;

enum{
    kFSInvalidObjectIteratorMovement= 0,
    kFSObjectEnter                  = 1,
    kFSObjectExit                   = 2,
};
```

Enumerator descriptions

`kFSInvalidObjectIteratorMovement`
: This is an invalid movement option.

`kFSObjectEnter`
: Enter the container most recently returned by a previous iteration.

`kFSObjectExit`
: Exit the container you are inside. If the outermost and current scope are defined as the same object, and you exit the current scope, your outermost scope is also redefined to the new current scope object. That's when you get the `E_ExitIteratorScope` result code that tells you that your outermost scope has expanded. You can continue using the iterator.

File Manager Functions

This section describes the functions in the File Manager application programming interface header file (`FileManager.h`), not the more complex functions in the File Manager SPI header file (`FileManagerSPI.h`). Occasionally, there are references to SPI functions, but you do not need them to accomplish the basic File Manager tasks. The SPI functions will be described in a later document.

Using File System Object References

This section describes the File Manager functions that explicitly handle file system object references as opposed to the objects themselves. Most File Manager functions use object references in one way or another, some implicitly creating object references and others using object references to identify objects.

You can clone, register, and dispose of file system object references. You can also obtain object references for objects within containers and for the containers themselves.

Note
Whenever a function returns an file system object reference, it implicitly creates an object reference that you are responsible for disposing of later. ◆

FSObjectCreateRef

Gets a file system object reference for a named object within a container.

```
OSStatus FSObjectCreateRef(
                    FSObjectRef container_i,
                    ConstFSName objectName_i,
                    FSObjectRef* objectRef_o);
```

`container_i`	The file system object reference for the object that contains the target object. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66), `FSObjectGetContainerRef` (page 2-68), and `FSObjectIterateOnce` (page 2-105).
`objectName_i`	The name of the target object.
`objectRef_o`	A pointer to the file system object reference for the target object. On output, the function returns this object reference. When you are done using this object reference, you are responsible for disposing of it.
function result	A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

You can use this function when you know the container and the name of an object, but not the object's object reference.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectGetContainerRef

Gets the file system object reference for the container of a given object.

```
OSStatus FSObjectGetContainerRef(
                    FSObjectRef objectRef_i,
                    FSObjectRef* containerRef_o);
```

`objectRef_i` The file system object reference for the object in question. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66), `FSVolumeGetInformation` (page 2-72), and `FSObjectIterateOnce` (page 2-105).

`containerRef_o`
A pointer to the file system object reference for the target container object. On output, the function returns this object reference. When you are done using this object reference, you are responsible for disposing of it.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectRefClone

Clones a file system object reference.

```
OSStatus FSObjectRefClone(
                    FSObjectRef object_t,
                    FSObjectRef* clone_o);
```

`object_t` The file system object reference to be copied. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66), `FSObjectGetContainerRef` (page 2-68), and `FSObjectIterateOnce` (page 2-105).

`clone_o` On output, a pointer to the cloned file system object reference, a copy of `object_t`. When you are done using the copied object reference, you are responsible for disposing of it.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

This function copies an object reference for use within the same kernel process. There are now *two* object references that your process needs to dispose of: the original object reference and the new copy.

You typically clone an object reference to balance a future call to the `FSObjectRefDispose` function (page 2-74).

Some examples where you might use this function:

- You have a function in your code that takes a file system object reference as input, does something to the object, and disposes of it. If you wanted to use that object reference after you called that function, you would need to clone it first. In this way, the function can safely dispose of one of the object references and you can continue to use the other copy until it disposes of it.
- An object-oriented programming example is a situation where you want to logically think of file system object references as being their own objects or as embedded in some other object. In the constructor for an object you would clone the object reference so that while that object exists you always have a usable object reference, yet creator of the object can dispose of the object reference whenever it likes. Then in the destructor for that object, you

would dispose of the object reference. You can think of it basically as two separate entities within the same program that both have their own interest in this object reference, and the clone gives you the opportunity to make that other copy so you dispose of them independently.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To copy an object reference for another process to use, you need to register the object reference with the `FSObjectRefRegister` function (page 2-70).

FSObjectRefRegister

Registers a file system object reference for another process.

```
OSStatus FSObjectRefRegister(
                    FSObjectRef senderObject_t,
                    KernelProcessID receiverPid_i);
```

`senderObject_t`

The file system object reference that is being registered. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66) and `FSObjectIterateOnce` (page 2-105). The receiver process is responsible for disposing of its copy of this object reference.

`receiverPid_i`

The receiver process–that is, other kernel process that will be using the registered file system object reference. When the receiver process is done using the registered object reference, it is responsible for disposing of the reference. In effect, the File Manager acts as if the `senderObject_t` object reference is returned to the `receiverPid_i` process.

function result A result code. See "File Manager Result Codes" (page 1-50) for a list of the result codes the File Manager can return.

DISCUSSION

This function is similar to the `FSObjectRefClone` function in that it copies a file system object reference, but it provides an object reference that is valid for a different kernel process.

This function is especially useful for the Code Fragment Manager (CFM) when it launches an application. The application is going to come up in a different process, so it will need an object reference that is valid in that process.

You might use this call if you have several processes where one process, typically a server of some kind, obtains object references for use by other processes, typically clients of that server. If the receiver process does not actually directly call the File Manager with that object reference, then you don't need to register the object reference to that process.

It would also be possible to have the server process make all of the calls to the File Manager. Object references can still be passed between client and server, but if the clients never use the references directly, then there is no need to register the references to those clients.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To copy an object reference for the same process to use, you need to clone the object reference with the `FSObjectRefClone` function (page 2-69).

FSVolumeGetInformation

Gets a volume's file system object reference.

```
OSStatus FSVolumeGetInformation(
                    FSVolumeObjID volume_t,
                    FSObjectRef* object_o);
```

`volume_t` The volume's object ID. You can use the constant `kFStheBootVolumeObject` to get the boot volume's object ID.

`object_o` A pointer to the volume's file system object reference. On output, the function returns this object reference. When you are done using this object reference, you are responsible for disposing of it.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSVolumeSetGetInformation

Gets a volume set's file system object reference.

```
OSStatus FSVolumeSetGetInformation(
                    FSVolumeSetObjID volumeSet_t,
                    Boolean* includesBootVolume_o,
                    FSObjectRef* object_o);
```

`volumeSet_t` The volume set's object ID. You can use the constant `kFStheBootVolumeSetObject` to get the boot volume set's object ID.

`includesBootVolume_o`
This has a value of `true` if the volume set includes the boot volume. Currently, the boot volume set is the only permitted volume set, so this should always be `true`.

`object_o` A pointer to the volume set's file system object reference. On output, the function returns this object reference. When you are done using this object reference, you are responsible for disposing of it.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectRefDispose

Disposes of a file system object reference.

```
OSStatus FSObjectRefDispose (FSObjectRef object_t);
```

`object_t` The file system object reference to be disposed. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66) and `FSObjectIterateOnce` (page 2-105).

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

A process must dispose of all file system object references returned to it.

Object references can be returned as explicit output parameters or as properties. If a reference is returned several times for a given object, it must be disposed of separately for each time it was returned.

When all references to a given object are disposed of, the File Manager disposes of any resources it allocated in order to operate on that object. The File Manager automatically disposes of all references for a process when the process terminates.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

For references returned as properties (especially when iterating over multiple objects), the more complex `FSObjectRefDisposeBulk` SPI function may be more convenient. See the `FileManagerSPI.h` header file for details of this function.

Using File System Objects

This section describes the File Manager functions that explicitly handle objects themselves as opposed to their file system object references. In order to use these functions, you need to know an object's object reference, but these functions do not return object references as output. You can move, rename, exchange, flush, and delete objects.

FSObjectRename

Renames a file or folder.

```
OSStatus FSObjectRename(
                    FSObjectRef sourceObjectRef_t,
                    ConstFSName newObjectName_i );
```

`sourceObjectRef_t`
The file system object reference of the file or folder to rename. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66), `FSObjectGetContainerRef` (page 2-68), and `FSObjectIterateOnce` (page 2-105).

`newObjectName_i`
The new name for the file or folder.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return. If another file or folder already exists in the new parent container with the same name as the renamed object's new name, the function returns the `E_DuplicateName` result code.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectMoveRename

Moves and renames a file or folder.

```
OSStatus FSObjectMoveRename(
                    FSObjectRef sourceObjectRef_t,
                    FSObjectRef destContainerRef_i,
                    ConstFSName newObjectName_i);
```

`sourceObjectRef_t`
The file system object reference of the file or folder in question. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66), `FSObjectGetContainerRef` (page 2-68), and `FSObjectIterateOnce` (page 2-105).

`destContainerRef_i`
The file system object reference of the container where the moved object is to be placed.

`newObjectName_i`
The new name for the moved file or folder. This is an optional parameter; pass a value of `NULL` if you don't want to change the name.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return. If another file or folder already exists in the new parent container that has the same name as the object or as the object's new name, the function returns the `E_DuplicateName` result code.

DISCUSSION

If you provided a new name for the file or folder object in the `newObjectName_i` parameter, this function also renames the object. If you don't want to change the name, pass a value of `NULL` for the `newObjectName_i` parameter.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectExchange

Exchanges the properties of two objects.

```
OSStatus FSObjectExchange(
                    FSObjectRef object1_i,
                    FSObjectRef object2_i);
```

`object1_i` The file system object reference of one object. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66) and `FSObjectIterateOnce` (page 2-105).

`object2_i` The file system object reference of the object to exchange it with.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

This call is used to allow a "safe save" that preserves an object's persistent reference. For example, you might want to save an updated set of properties to a temporary object so that any errors while saving result in the original object being unchanged; but, you also want the object's persistent reference to remain unchanged so that aliases still work.

What you do is create a second object somewhere (in a temporary folder, for example). Write out its properties, both unchanged and changed, to the second object. When done saving, call the `FSObjectExchange` function; the fork properties and the modification date of the two objects are swapped in such a way that the original object has the new properties but retains its old persistent reference.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectFlush

Flushes any data cached for an object.

```
OSStatus FSObjectFlush (FSObjectRef object_t);
```

`object_t` The file system object reference of the object to be flushed. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66) and `FSObjectIterateOnce` (page 2-105)..

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

If the object is a file, then any data written to it by a stream or backing store is written by the File Manager to its underlying device. If the object is a volume, then the volume-level metadata for that volume is flushed.

Any changes to the object's properties are flushed, regardless of the object's type. Data about the object, or contained in the object, can still reside in the File Manager's caches, but any changes are written out.

Note that the underlying device's driver, or the device itself, may cache some data, so the File Manager cannot guarantee that all data has actually been written to the underlying media.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectDelete

Deletes an object.

```
OSStatus FSObjectDelete (FSObjectRef object_t);
```

`object_t` The object to be deleted. You can get a file system object reference from several functions, such as `FSObjectCreateRef` (page 2-66) and `FSObjectIterateOnce` (page 2-105).

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

This function does not dispose of object's file system object reference; you must still dispose it yourself. Further attempts to use the object reference return an `E_ObjectNotFound` result code.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Creating Files and Folders

FSFileCreate

Creates a new named file.

```
OSStatus FSFileCreate(
                    FSObjectRef containerRef_t,
                    ConstFSName fileName_i,
                    OSType fileCreator_i,
                    OSType fileType_i,
                    FSObjectRef* fileRef_o);
```

`containerRef_t`
The file system object reference of the object to contain the new file. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66) and `FSObjectIterateOnce` (page 2-105).

`fileName_i` The name of the new file as a persistent text object.

`fileCreator_i`
The new file's Finder creator.

`fileType_i` The new file's Finder file type.

`fileRef_o` A pointer to the file system object reference of the new file. On output, the function returns this object reference. When you are done using this object reference, you are responsible for disposing of it.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return. If another file already exists in that container with the same name, creator, and type, the function returns the `E_DuplicateName` result code.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSFolderCreate

Create a new named folder within a specified folder.

```
OSStatus FSFolderCreate(
                    FSObjectRef containerRef_t,
                    ConstFSName folderName_i,
                    FSObjectRef* folderRef_o);
```

`containerRef_t`
The file system object reference of the object to contain the new folder. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66) and `FSObjectIterateOnce` (page 2-105).

`folderName_i` The name of the new folder as a persistent text object.

`folderRef_o`	A pointer to the file system object reference of the new folder. On output, the function returns this object reference. When you are done using this object reference, you are responsible for disposing of it.
function result	A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return. If another folder already exists with the same name in that container, the function returns the `E_DuplicateName` result code.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Getting and Setting Properties

FSObjectGetOneProperty

Gets a simple property attribute of an object.

```
OSStatus FSObjectGetOneProperty(
                    FSObjectRef objectRef_t,
                    const FSProperty* property_i,
                    ByteCount propertySize_i,
                    LogicalAddress property_o);
```

`objectRef_t`	The file system object reference of the object in question. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66) and `FSObjectIterateOnce` (page 2-105).
`property_i`	A pointer to the property attribute to get.
`propertySize_i`	The size (in bytes) of the property attribute data specified by the `property_o` parameter.
`property_o`	The address of a buffer in which to return the requested property data. You are responsible for allocating an adequately large buffer.
function result	A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

You can use this function to get the value of a simple property as well as any other attributes of all properties such as size, type, and state. You cannot use this function to get the value attribute of a fork property; for this you must use stream or memory-mapped file access functions, such as `FSStreamSimpleRead` (page 2-97) and `FSMappedFileOpen` (page 2-99).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectSetOneProperty

Sets a property attribute of an object.

```
OSStatus FSObjectSetOneProperty(
                    FSObjectRef objectRef_t,
                    const FSProperty* property_i,
                    ByteCount propertySize_i,
                    ConstLogicalAddress propertyData_i);
```

`objectRef_t` The file system object reference of the object in question. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66) and `FSObjectIterateOnce` (page 2-105).

`property_i` A pointer to the property attribute to set. The attribute to get is specified by `property_i->attribute`.

`propertySize_i`
The size (in bytes) of the property attribute data specified by the `propertyData_i` parameter.

`propertyData_i`
The address of the buffer that holds the new property data.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

You can use this function to set the value of a simple property as well as any other attributes of all properties such as size, type, and state. You *cannot* use this function to set the value attribute of a fork property; for this attribute, you must use stream or memory-mapped file access functions, such as `FSStreamSimpleWrite` (page 2-98) and `FSMappedFileOpen` (page 2-99).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Getting File System Object Information

This section describes the functions that return an object information structure, as defined by the `FSObjectInformation` data type (page 2-19), which provides separate information structures for files, folders, and volumes. These structures contain a predefined set of aggregate property data such as the object's option flags and Finder information.

FSObjectGetInformation

Gets an object information structure for a file or folder.

```
OSStatus FSObjectGetInformation(
                    FSObjectRef objectRef_t,
                    FSInfoVersion infoVersion_i,
                    FSObjectInformation* objectInfo_o,
                    FSName objectName_o);
```

`objectRef_t` The file system object reference of the object in question. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66) and `FSObjectIterateOnce` (page 2-105).

`infoVersion_i`

The version of the object information structure specified by the `objectInfo_o` parameter. Use the constant

	`kFSInfoCurrentReleasedVersion` for this parameter to specify the latest version of the object information structure. This parameter is ignored if `objectInfo_o` is omitted.
`objectInfo_o`	A pointer to the object information structure being returned. This is an optional parameter; pass a null pointer if you don't want the File Manager to provide this structure on output.
`objectName_o`	The name of the specified object. If you specify this, it must reference a preinitialized persistent text object of sufficient size to contain the object's name. This is an optional parameter; pass a value of `NULL` if you don't want the File Manager to provide the name on output.
function result	A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectGetVolumeInformation

Gets an object information structure for a volume.

```
OSStatus FSObjectGetVolumeInformation(
                    FSObjectRef volumeItemRef_t,
                    FSInfoVersion infoVersion_i,
                    FSObjectInformation* volumeInfo_o,
                    FSObjectRef* volumeObjectRef_o,
                    FSName volumeName_o);
```

`volumeItemRef_t`
The file system object reference of a volume item such as a file or folder or the object reference for the volume itself. You can get an object reference from several functions, such as `FSObjectCreateRef` (page 2-66), `FSObjectGetContainerRef` (page 2-68), and and `FSObjectIterateOnce` (page 2-105).

`infoVersion_i`
The version of the object information structure specified by the `volumeInfo_o` parameter. Use the constant `kFSInfoCurrentReleasedVersion` for this parameter to specify the latest version of the object information structure. This parameter is ignored if `volumeInfo_o` is omitted.

`volumeInfo_o` A pointer to the object information structure being returned. This is an optional parameter; pass a null pointer if you don't want the File Manager to provide this structure on output.

`volumeObjectRef_o`
A pointer to the object reference of the volume. This is an optional parameter; pass a null pointer if you don't want the File Manager to provide this object reference on output.

`volumeName_o` The name of the specified volume. If you specify this, it must reference a preinitialized persistent text object of sufficient size to contain the object's name. This is an optional parameter; pass a value of `NULL` if you don't want the File Manager to provide the name on output.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

This function is functionally equivalent to the `FSObjectGetInformation` function when the target object reference is the volume's object reference.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Using Stream Access Methods

You must use stream or memory-mapped file access methods to get and set the value attribute of a fork property.

FSStreamOpen

Opens a file fork for stream access.

```
OSStatus FSStreamOpen(
                    FSObjectRef fileObjectRef_i,
                    FSForkType fork_i,
                    FSStreamObjID* stream_o);
```

`fileObjectRef_i`
The file system object reference of the file you wish to open.

`fork_i` The type of fork to open. The only allowable values are `kFSDataFork` and `kFSResourceFork`.

`stream_o` A pointer to the new stream ID. On output, the function returns the stream ID.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return. If the file is locked or is open on another stream with conflicting access constraints, then the function returns the `E_PermissionViolation` result code.

[••• Review query: Which error code is this? **•••]**

DISCUSSION

This function attempts to open the fork with exclusive read/write access.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To specify particular access constraints, the more complex `FSStreamOpenWithConstraints` SPI function may be more convenient. See the `FileManagerSPI.h` header file for details of this function.

FSStreamClose

Closes a stream.

```
OSStatus FSStreamClose (FSStreamObjID stream_t);
```

`stream_t` The object ID of the stream to be closed.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

This function closes a stream that had been previously opened with the `FSStreamOpen` function (or the `FSStreamOpenWithConstraints` SPI function).

Any data written to the stream is flushed, as with the `FSStreamFlush` function (page 2-91), before the stream is closed. The File Manager disposes of any resources allocated for use by this stream, releases any range locks, and makes invalid the stream ID specified by the `stream_t` parameter.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSStreamFlush

Flushes any data written to a stream.

```
OSStatus FSStreamFlush (FSStreamObjID stream_t);
```

`stream_t` The object ID of the stream to be flushed.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

Typically you would use this function when you want to continue to use a stream, keeping it open, but you want to make sure its contents are occasionally written out. (If you are finished with the stream, you would use the `FSStreamClose` function (page 2-90) instead.)

Stream data can still reside in the File Manager's caches, but any changes are written out by the File Manager. Note that the underlying device's driver, or the device itself, may cache some data, so the File Manager cannot guarantee that all data has actually been written to the underlying media.

Any volume-level data needed to access the stream is flushed, but other information about the object (such as its modification date) might not be flushed by this function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSStreamGetAbsoluteEOF

Gets the logical EOF of an open stream.

```
OSStatus FSStreamGetAbsoluteEOF(
                    FSStreamObjID stream_t,
                    FSOffset* currentEOF_o);
```

`stream_t` The object ID of the stream in question.

`currentEOF_o` A pointer to the stream's logical end-of-file (EOF). On output, the function returns the EOF.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSStreamSetAbsoluteEOF

Sets the logical EOF of an open stream.

```
OSStatus FSStreamSetAbsoluteEOF(
                    FSStreamObjID stream_t,
                    const FSOffset* eof_i);
```

`stream_t` The object ID of the stream in question.

`eof_i` A pointer to the stream's new logical end-of-file (EOF).

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

If there is not enough space on the volume to set the stream's EOF to the offset specified by the `eof_i` parameter, then the EOF is not changed and the function returns the `E_DiskFull` result code.

DISCUSSION

This function sets the logical end of file, which allows you to adjust the size of the fork. You could use this function to extend a fork before writing out additional data or you could use the function to shorten a fork from which you have just deleted data to release the unused space. For purposes of range locks, changing the EOF acts as a write between the old and new EOF.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSStreamGetMark

Gets a stream's current mark.

```
OSStatus FSStreamGetMark(
                    FSStreamObjID stream_t,
                    FSOffset* currentMark_o);
```

`stream_t` The object ID of the stream in question.

`currentMark_o`
A pointer to the stream's current mark–that is, the current position offset. On output, the function returns the current mark.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

This function returns the offset from the start of the file that would be equivalent to using a `FSForkPositionDescriptor` whose `positionOffset` is 0, and whose `positionMode` is `kFSFromMark`. A stream's mark is automatically set by the File Manager to the byte following the last read or write, or you can set it manually with the `FSStreamSetMark` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSStreamSetMark

Sets a stream's mark to a new position.

```
OSStatus FSStreamSetMark(
                    FSStreamObjID stream_t,
                    const FSForkPositionDescriptor* newPosition_i,
                    FSStreamSetMarkOptions options_i,
                    FSOffset* originalMark_o,
                    FSOffset* currentMark_o);
```

`stream_t` The object ID of the stream in question.

`newPosition_i`
The new position of the stream's mark.

`options_i` The options set for this function. Currently, the only valid options are 0 and `kFSMarkPinToEOF`.

`originalMark_o`
A pointer to the stream's previous mark–that is, the offset that is being changed by this function. On output, the function returns the previous mark.

`currentMark_o`
A pointer to the stream's new current mark–that is, the current position offset relative to the start of the stream. On output, the function returns the current mark.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return. *Otherwise*, an error is returned.

If the `options_i` parameter is set to the `kFSMarkPinToEOF` constant and the new mark position specified by `newPosition_i` exceeds the stream's current EOF, then the function sets the mark to the stream's EOF and returns the `E_EndOfFileErr` result code. Otherwise, an `E_PosOutOfRange` result code is returned. The mark can never be set past the end of the stream.

DISCUSSION

A stream's mark is usually used for sequential access to a stream or to establish a position relative to the ending position of the last operation on a stream. This function lets you explicitly set the mark for future operations that will operate relative to the current mark. A situation where this is especially useful is when you are working with a large or complex structure: you can set the mark to accomodate entire structures, so that the next read or write is positioned at the beginning of the next structure.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSStreamSimpleRead

Reads data from an open stream.

```
OSStatus FSStreamSimpleRead(
                    FSStreamObjID stream_t,
                    ByteCount requestLength_i,
                    const FSForkPositionDescriptor* position_i,
                    LogicalAddress data_o,
                    ByteCount* actualLength_o,
                    FSOffset* currentMark_o);
```

`stream_t` The object ID of the stream in question.

`requestLength_i`
The number of bytes to read from the stream.

`position_i` A pointer to the starting position for the read operation.

`data_o` The address where the requested data is to be returned. On output, the function returns the data.

`actualLength_o`
A pointer to the actual number of bytes read from the stream. This is an optional parameter; pass a null pointer for this parameter when you don't want the File Manager to provide this length value on output.

`currentMark_o`
A pointer to the stream's mark position after the read operation. This is an optional parameter; pass a null pointer for this parameter when you don't want the File Manager to provide this mark value to you on output. The stream mark is, however, always positioned after the last byte read by a successful `FSStreamSimpleRead` function.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return. If you try to read beyond the stream's EOF, then the function sets the mark to the EOF, fills in the `actualLength_o` parameter with the number of bytes actually read, and returns the `E_EndOfFileErr` result code. If you are already at the EOF, then the `actualLength_o` parameter is set to 0, and you get the `E_EndOfFileErr` result code.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSStreamSimpleWrite

Writes data to an open stream.

```
OSStatus FSStreamSimpleWrite(
                    FSStreamObjID stream_t,
                    ByteCount requestLength_i,
                    ConstLogicalAddress data_i,
                    const FSForkPositionDescriptor* position_i,
                    ByteCount* actualLength_o,
                    FSOffset* currentMark_o);
```

`stream_t` The object ID of the stream in question.

`requestLength_i`
The number of bytes to write to the stream.

`data_i` The address of a buffer containing the data to write.

`position_i` A pointer to the starting position for the write operation.

`actualLength_o`
A pointer to the actual number of bytes written to the stream. This is an optional parameter; pass a null pointer for this parameter when you don't want the File Manager to provide this length value on output.

`currentMark_o`
A pointer to the stream's mark position after the write operation. This is an optional parameter; pass a null pointer for

this parameter when you don't want the File Manager to provide this length value to you on output. The stream mark is, however, always positioned after the last byte written by a successful `FSStreamSimpleWrite` function.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return. If you try to write beyond the stream's EOF, then the function moves the EOF to the byte following the last written byte.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Using Memory-Mapped File Access Methods

You must use stream or memory-mapped file access methods to get and set the value attribute of a fork property.

FSMappedFileOpen

Opens a file fork for memory-mapped access.

```
OSStatus FSMappedFileOpen(
                    FSObjectRef fileObjectRef_t,
                    FSForkType fork_i,
                    FSBackingStoreObjID* backingStore_o);
```

`fileObjectRef_t`
The file system object reference of the file to open.

`fork_i` The type of fork to open. The only allowable values are `kFSDataFork` and `kFSResourceFork`.

`backingStore_o`
A pointer to the object ID of the backing store used to access the fork. On output, the function returns the object ID. This is the value you pass to the kernel `CreateArea` function as its `backingObject` parameter in order to create memory for your backing store object.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.If the file is locked or is open on another stream with conflicting access constraints, then the function returns the `E_PermissionViolation` result code.

DISCUSSION

This function attempts to open the fork with exclusive read/write access.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To specify particular access constraints, the more complex `FSMappedFileOpenWithConstraints` SPI function may be more convenient. See the `FileManagerSPI.h` header file for details of this function.

FSMappedFileClose

Closes an access path to a file used for backing store.

```
OSStatus FSMappedFileClose (FSBackingStoreObjID backingStore_t);
```

`backingStore_t`
The object ID of the backing store to be closed.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

This function closes a backing store that had been opened with the `FSMappedFileOpen` function (or the `FSMappedFileOpenWithConstraints` SPI function).

Any data written to this backing store by writing to pages backed by this store is flushed (that is, written out by the File Manager) before the backing store is closed. The File Manager disposes of any resources allocated for use by this backing store and makes invalid the the backing store ID specified by the `backingStore_t` parameter.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To open a file with particular access constraints, you need to use the more complex `FSMappedFileOpenWithConstraints` SPI function. See the `FileManagerSPI.h` header file for details of this function.

FSMappedFileGetAbsoluteEOF

Gets the EOF of the fork being accessed by a backing store.

```
OSStatus FSMappedFileGetAbsoluteEOF(
                    FSBackingStoreObjID backingStore_t,
                    FSOffset* currentEOF_o);
```

`backingStore_t`
The object ID of the backing store used to access the fork.

`currentEOF_O` A pointer to the fork's current EOF–that is, the fork's size (in bytes). On output, the function returns the EOF.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

Since memory-mapped file access to a fork is accomplished by directly accessing memory pages, the virtual memory system must read and write entire pages. If any data on the last page is modified, the entire page is written, resulting in the fork size being rounded up to a multiple of a page size. This also true for access to pages beyond the fork's EOF.

You can set the fork's EOF implicitly by writing to backed pages or explicitly by using the `FSMappedFileSetEOF` function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSMappedFileSetAbsoluteEOF

Sets the EOF of the fork being accessed by a backing store.

```
OSStatus FSMappedFileSetAbsoluteEOF(
                    FSBackingStoreObjID backingStore_t,
                    const FSOffset* eof_i,
                    FSOffset* currentEOF_o);
```

`backingStore_t`
The object ID of the backing store used to access the fork.

`eof_i` A pointer to the fork's new EOF.

`currentEOF_o` A pointer to the fork's current EOF–that is, the fork's size (in bytes). On output, the function returns the EOF.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

Since access to a fork via a backing store (that is, memory-mapped file access) is accomplished by directly accessing memory pages, the virtual memory system must read and write entire pages. If anything on the last page is modified, the entire page is written, resulting in the fork size being rounded up to a multiple of a page size. Similarly for access to pages beyond the fork's EOF.

This call would typically be used when a fork has been memory mapped to enable convenient access to a file's data structures as if it were completely in memory. You would make all changes to the data structures, then use this call to indicate the number of bytes that are valid and should be written to the fork.

This call allows the EOF to be explicitly set for a fork being accessed via a backing store. Any data beyond the EOF is not actually written to the fork and the File Manager has no way to detect access to pages beyond the EOF.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Iterating Over File System Objects

This section describes the File Manager functions that create and use object iterators, that change an iterator's current scope, and that restart and dispose of iterators.

FSObjectIteratorCreate

Creates an iterator.

```
OSStatus FSObjectIteratorCreate(
                    FSObjectRef outermostScope_t,
                    FSObjectIteratorCreationOptions options_i,
                    FSObjectIteratorObjID* iterator_o);
```

`outermostScope_t`
The object reference of the object that is the outermost scope. Initially, the outermost and current scopes are set to the same object. The outermost and current scope objects must be objects that can contain other objects–that is, the universe, a volume set, a volume, or a folder, but not a file.

`options_i`
A series of options that control how an iterator behaves: which kinds of objects it returns and whether or not it traverses embedded containers. These options are defined in the `FSObjectIteratorCreationOptions` enumeration (page 2-63).

`iterator_o`	A pointer to the object ID of the iterator. On output, the function returns the object ID. When you have finished using an iterator, call the `FSObjectIteratorDispose` function to dispose of it.
function result	A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

When you create an iterator, it is not positioned on any object, although it is inside its current scope (and, since they are initially the same, it is inside its outermost scope also). It is in a state of `kFSIteratorSOI` (start of iteration) meaning that all objects in the current scope have yet to be returned.

There are several options that you can use to determine an iterator's behavior when it iterates. These are defined by the `FSObjectIteratorCreationOptions` enumeration (page 2-63).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectIterateOnce

Iterates once to the next object.

```
OSStatus FSObjectIterateOnce(
                    FSObjectIteratorObjID iterator_t,
                    FSInfoVersion objectInfoVersion_i,
```

```
                    FSObjectInformation* objectInfo_o,
                    FSObjectRef* objectRef_o,
                    FSName objectName_o);
```

`iterator_t` The object ID of the iterator.

`objectInfoVersion_i`
The version of the object information structure specified by the `objectInfo_o` parameter. Use the constant `kFSInfoCurrentReleasedVersion` for this parameter to specify the latest version of the object information structure. This parameter is ignored if `objectInfo_o` is omitted.

`objectInfo_o` A pointer to the object information structure being returned. This is an optional parameter; pass a null pointer if you don't want the File Manager to return this structure on output.

`objectRef_o` A pointer to the file system object reference of the current iterator object. This is an optional parameter; pass a null pointer if you don't want the File Manager to return this structure on output.

`objectName_o` The name of the specified object. If you specify this, it must reference a preinitialized persistent text object of sufficient size to contain the object's name. This is an optional parameter; pass a value of `NULL` if you don't want the File Manager to return this name on output.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return. If the iterator has already returned all the appropriate objects in its current scope, then the function returns the `E_EndOfIteration` result code.
If, however, any container in the scope stack (between the outermost and current scopes has been moved or deleted, you get the `E_IteratorScopeException` result: You are no longer where you thought you were. You cannot continue to use the iterator until you recreate it with the `FSObjectIteratorRecreate` SPI function or dispose of it with the `FSObjectIteratorDispose` function (page 2-110).

DISCUSSION

This function attempts to return information for the next object that meets all the currently established iteration criteria: the options you set when you created the iterator with the `FSObjectIteratorCreate` function (page 2-104), and any changes you have made to the current scope with the `FSObjectIteratorChangeCurrentScope` function (page 2-107).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectIteratorChangeCurrentScope

Changes an object iterator's current scope.

```
OSStatus FSObjectIteratorChangeCurrentScope(
                    FSObjectIteratorObjID iterator_t,
                    FSObjectIteratorMovement movement_i);
```

`iterator_t` The object ID of the iterator.

`movement_i` The direction of movement for the iterator: into or out of a container. These are defined in the `FSObjectIteratorMovement` enumeration (page 2-65).

If the movement is set to `kFSObjectEnter`, then the iterator will be positioned inside the container object last returned by the most recent iteration, but not on any particular object. In fact, the container could be empty. That container object becomes the

new current scope of the iterator and the iterator is put into a state of `kFSIteratorSOI` (start of iteration) meaning that all objects in the current scope have yet to be returned.

If the movement is set to `kFSObjectExit`, then the current scope moves out of the current container to become the object that contains this container. That is, if folder A (previous current scope) is in inside the folder WrapsA, and you want to exit the current scope, then the WrapsA folder becomes the new current scope.

If the current scope and the outermost scope were the same before the iteration, then the outermost scope also changes to the new current scope and the function returns the `E_ExitIteratorScope` result code so that you realize your next iteration will be outside the scope that you used to create the iterator. The iterator remains usable.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

If any object in the scope stack (that is, any object between the outermost scope and the current scope) is moved, the iterator is invalidated and the function returns the `E_IteratorScopeException` result code until it has been explicitly fixed (by the SPI function `FSObjectIteratorRecreate`) or disposed of by the `FSObjectIteratorDispose` function (page 2-110). This function adds or removes objects from the scope stack.

SPECIAL CONSIDERATIONS

You cannot use this function if you created the iterator with the `kFSTraverseEmbeddedContainers` option.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To fix an invalidated iterator, you may need to use the SPI function `FSObjectIteratorRecreate`. See the `FileManagerSPI.h` header file for details of this function.

FSObjectIteratorRestart

Restarts an object iterator.

```
OSStatus FSObjectIteratorRestart (FSObjectIteratorObjID iterator_t);
```

`iterator_t` The object ID of the iterator.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

The iterator is put into an object iteration state of `kFSIteratorSOI` (start of iteration), meaning that all objects in the current scope have yet to be returned. The iterator is not positioned on any object; it is positioned before the first object in the scope that was used to create the iterator (that is, the outermost scope).

You use this function to completely restart iteration within the current scope, ignoring any state information about objects previously returned in the current

scope. The outermost scope is not affected. State information about which objects have been returned from scopes outside the current scope is unchanged.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSObjectIteratorDispose

Disposes of an object iterator.

```
OSStatus FSObjectIteratorDispose (FSObjectIteratorObjID iterator_t);
```

`iterator_t` The object ID of the iterator.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

This functions prompts the File Manager to dispose of the iterator and release any resources allocated to it. Further attempts to use the iterator will result in an error.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Cross Referencing Object References and FSSpec File Specifications

FSObjectRefGetFSSpec

Gets an `FSSpec` file specification for a given file system object reference.

```
OSStatus FSObjectRefGetFSSpec(
                    FSObjectRef object_t,
                    FSSpec* fSSpec_o);
```

`object_t` A pointer to the file system object reference for which you wish to obtain a `FSSpec` file specification.

`fSSpec_o` A pointer to the `FSSpec` file specification, suitable for use with the System 7 File Manager application programming interface, that corresponds to the given file system object reference. On output, the function returns the `FSSpec` file specification.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

You can use this function if your code uses both the System 7 File Manager application programming interface and the Mac OS 8 FileManager (or has clients that use both types of file system software). For example, if you have a

piece of code that has a function that still uses the `FSSpec` data type, but that has been converted internally to use the file system object references; you can use this function to produce an `FSSpec` data type as an output for the preexisting function.

Note
You are strongly recommended to provide an application programming interface that uses file system object references. ▲

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

FSSpecGetFSObjectRef

Gets a file system object reference for an object corresponding to an `FSSpec` data type.

```
OSStatus FSSpecGetFSObjectRef(
                    const FSSpec* theFSSpec_t,
                    FSObjectRef* theObject_o);
```

`theFSSpec_t` A pointer to the `FSSpec` file specification for which you wish to obtain a file system object reference. The `FSSpec` file specification must be fully normalized; that is, it must not contain any working directories.

`theObject_o`	A pointer to the file system object reference for the File Manager object that corresponds to the file or directory specified by the `FSSpec` file specification. On output, the function returns the object reference. When you are done using the object reference, you are responsible for disposing of it.
function result	A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

You can use this function if your code uses both the System 7 File Manager application programming interface and the Mac OS 8 File Manager (or has clients that use both types of file system software). For example, if you have a piece of code that has a function that still uses the `FSSpec` data type, but that has been converted internally to use the object references; you can use this function to convert an `FSSpec` data type into an object reference for use internally.

Note
You are strongly recommended to provide an application programming interface that uses file system object references. ▲

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Resolving Pathnames

FSPathnameResolve

Gets the file system object reference for an object with a given pathname.

```
OSStatus FSPathnameResolve(
                    FSObjectRef container_i,
                    const char* path_i,
                    ByteCount pathLength_i,
                    FSPathnameType pathType_i,
                    FSObjectRef* objectRef_o);
```

`container_i` The object reference for the path's parent container. If the `pathType_i` parameter is set to the constant `kFSHFSPath` and the `path_i` parameter is a full HFS path, then the object identified by the `container_i` parameter is ignored.

`path_i` A pointer to the partial pathname identifying where the container is located. This is a list of names separated by the delimiters appropriate for the path type specified in the `pathType_i` parameter. This function follows this path down through every container in the path until it comes to the uniquely identified object within the last container.

`pathLength_i` The length (in bytes) of the pathname.

`pathType_i` The type of path to use in constructing the input path, such as HFS. This tells the File Manager how to interpret a pathname. For example, HFS paths use colons (:) as delimiters, UNIX paths use slashes (/), and DOS paths use backwards slashes (\). The permitted values are defined by the `FSPathnameType` structure (page 2-16).

`objectRef_o` A pointer to the file system object reference for the object ultimately identified by the container-pathname combination. On output, the function returns the object reference. When you are done using the object reference, you are responsible for disposing of it.

function result A result code. See "File Manager Result Codes" (page 2-115) for a list of the result codes the File Manager can return.

DISCUSSION

This function is useful for porting existing programs or writing new programs that need to be able to handle different notations and delimiters for pathnames. Note that the File Manager does not support drive specifiers such as C: in DOS path and root specifiers in UNIX paths. In such cases, this function starts at the parent container specified in the `container_i` parameter and parses the indicated path from there.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

File Manager Result Codes

The File Manager returns many different result codes. This section provides the current set of result codes. The list is subject to change is later releases.

Basic Error Types

```
enum{
    FSFilesAPIErrorBias = 0xF4000000L,
                            // the upper short that identifies a file system error
    FSErrorBias         = 0xF5000000L,
                            // the upper short that identifies a file system error
```

```
    FSErrorBiasMask     = 0xFF000000L,
                            // how to discriminate non-FileManager errors
    FSAgentErrorBias    = 0xF6000000L
                            // the upper short that identifies an agent error
};
```

Error Mask Types

```
enum{
    FSFilesAPIErrorMask     = FSFilesAPIErrorBias | 0xFFFF,
                                // how to convert from old error codes to new ones
    FSErrorMask             = FSErrorBias | 0xFFFF,
                                // how to convert from old error codes to new ones
    FSAgentErrorBiasMask    = FSAgentErrorBias | 0xFFFF
                                // how to discriminate non-agent errors
};
```

Mac OS-Aliased Exceptions

```
enum {
    E_NoErr                 = (noErr&FSErrorMask),
    E_NoError               = E_NoErr,                  // alias for noErr
    E_ParamErr              = (paramErr&FSErrorMask),
                                    // alias for paramErr in Errors.h
    E_FolderFulErr          = (dirFulErr&FSErrorMask),  // folder full
    E_DirFulErr             = E_FolderFulErr,           // directory full
    E_DskFulErr             = (dskFulErr&FSErrorMask),  // disk full
    E_DiskFull              = E_DskFulErr,
    E_NSVErr                = (nsvErr&FSErrorMask),     // no such volume
    E_VolumeNotFound        = E_NSVErr,
    E_IOErr                 = (ioErr&FSErrorMask),      // I/O Err
    E_BdNamErr              = (bdNamErr&FSErrorMask),
                                    // there may be no bad names in the final system!
    E_BadName               = E_BdNamErr,
    E_FnOpnErr              = (fnOpnErr&FSErrorMask),   // file not open
    E_FileNotOpen           = E_FnOpnErr,
    E_EofErr                = (eofErr&FSErrorMask),     // end of file
```

```
E_EndOfFileErr          = E_EofErr,
E_PosErr                = (posErr&FSErrorMask),
                                // tried to position to before start of file (r/w)
E_PosOutOfRange         = E_PosErr,
                                // position is invalid (before start of file,
                                // or out of range for filesystem)
E_MFulErr               = (mFulErr&FSErrorMask),
                                // memory full (open) or file won't fit (load)
E_MemFullErr            = E_MFulErr,
E_TMFOErr               = (tmfoErr&FSErrorMask),    // too many files open
E_FnfErr                = (fnfErr&FSErrorMask),     // file not found
E_FileNotFound          = E_FnfErr,
E_WPrErr                = (wPrErr&FSErrorMask),     // diskette is write protected.
E_WriteProtected        = E_WPrErr,
E_FLckdErr              = (fLckdErr&FSErrorMask),   // file is locked
E_FileLocked            = E_FLckdErr,
E_VLckdErr              = (vLckdErr&FSErrorMask),   // volume is locked
E_VolumeLocked          = E_VLckdErr,
E_FBsyErr               = (fBsyErr&FSErrorMask),    // file is busy (delete)
E_FileInUse             = E_FBsyErr,
E_DupFNErr              = (dupFNErr&FSErrorMask),   // duplicate filename (rename)
E_DuplicateName         = E_DupFNErr,
E_OpWrErr               = (opWrErr&FSErrorMask),
                            // file already open with with write permission
E_WriteAccessDenied     = E_OpWrErr,
E_RfNumErr              = (rfNumErr&FSErrorMask),   // refnum Err
E_BadObjectID           = E_RfNumErr,
E_GfpErr                = (gfpErr&FSErrorMask),     // get file position Err
E_GetFilePosition       = E_GfpErr,
E_VolOffLinErr          = (volOffLinErr&FSErrorMask),// volume not on line Err
E_VolumeOffline         = E_VolOffLinErr,
E_PermErr               = (permErr&FSErrorMask),    // permissions Err
E_PermissionViolation   = E_PermErr,
E_VolOnLinErr           = (volOnLinErr&FSErrorMask),
                            // drive volume already on-line at MountVol
E_NSDrvErr              = (nsDrvErr&FSErrorMask),
                            // no such drive (tried to mount a bad drive num)
E_NoMacDskErr           = (noMacDskErr&FSErrorMask),
                            // not a Mac diskette (sig bytes are wrong)
E_ExtFSErr              = (extFSErr&FSErrorMask),
                            // volume in question belongs to an external fs
```

```
    E_FSRnErr               = (fsRnErr&FSErrorMask),    // file system internal Err
    E_BadMDBErr             = (badMDBErr&FSErrorMask),  // bad master directory block
    E_WrPermErr             = (wrPermErr&FSErrorMask),  // write permissions Err
    E_DirNFErr              = (dirNFErr&FSErrorMask),   // directory not found
    E_FolderNotFound        = E_DirNFErr,
    E_DirectoryNotFound     = E_FolderNotFound,
    E_TMWDOErr              = (tmwdoErr&FSErrorMask),   // no free WDCB available
    E_BadMovErr             = (badMovErr&FSErrorMask),  // move into offspring Err
    E_IllegalMove           = E_BadMovErr,
    E_WrgVolTypErr          = (wrgVolTypErr&FSErrorMask),// wrong volume type Err
    E_VolGoneErr            = (volGoneErr&FSErrorMask),
                                // Server volume has been disconnected.
    E_FIDNotFound           = (fidNotFound&FSErrorMask),// no file thread exists.
    E_FIDExists             = (fidExists&FSErrorMask),  // file ID already exists
    E_NotAFileErr           = (notAFileErr&FSErrorMask),// non-file object specified
    E_DiffVolErr            = (diffVolErr&FSErrorMask), // files on different volumes
    E_CatChangedErr         = (catChangedErr&FSErrorMask),
                                // the catalog has been modified
    E_DesktopDamagedErr     = (desktopDamagedErr&FSErrorMask),
                                // desktop database files are corrupted
    E_SameFileErr           = (sameFileErr&FSErrorMask),
                                // can't exchange a file with itself
    E_BadFidErr             = (badFidErr&FSErrorMask),
                                // file ID is dangling or doesn't match the file number
    E_AfpItemNotFound       = (afpItemNotFound&FSErrorMask),// information not found
    E_AfpIconTypeError      = (afpIconTypeError&FSErrorMask),
                                // sizes of new icon and one it replaces don't match
};
```

General Exceptions - Sharable by Different Modules

```
enum{
    E_MissingParameter          = FSErrorBias | 0x0001,
                                // one or more mandatory parameters missing
    E_InvalidMsg                = FSErrorBias | 0x0002,// invalid message
//  reserved                    = FSErrorBias | 0x0003,
    E_InvalidForkType           = FSErrorBias | 0x0004,
                                        // fork type not recognized or not supported
    E_ForkInUse                 = FSErrorBias | 0x0005, // fork is currently in use
```

```
    E_PropertyNotFound          = FSErrorBias | 0x0006, // requested Property not found
    E_ObjectNotFound            = FSErrorBias | 0x0007, // requested Object not found
    E_MemoryFull                = E_MemFullErr,         // PoolAllocate returned nil
    E_NotAFileOrFolder          = FSErrorBias | 0x0008,
                                        // given object is not a file or a Folder
    E_TooManyParameters         = FSErrorBias | 0x0009,
                                        // one or another parameter, but not both
    E_PatternNotFound           = FSErrorBias | 0x000A,
                                        // couldn't find pattern in buffer
    E_BufferLength              = FSErrorBias | 0x000B, // illegal buffer size
    E_InvalidDirectoryNum       = FSErrorBias | 0x000C, // invalid Directory Number
    E_IllegalFileOperation      = FSErrorBias | 0x000D,
    E_IllegalFolderOperation    = FSErrorBias | 0x000E,
    E_IllegalVolumeOperation    = FSErrorBias | 0x000F,
    E_InvalidRelationship       = FSErrorBias | 0x0010,
                                    // relationship is invalid in context of operation
    E_UnknownRelationship       = FSErrorBias | 0x0011, // relationship is undefined
    E_DoesNotMatch              = FSErrorBias | 0x0012,
};
```

FSAgent Interface Exceptions

These result codes are used internally by volume-format plug-ins.

```
enum{
    E_NoGetBlockProc        = FSErrorBias | 0x0101,
    E_NoReleaseBlockProc    = FSErrorBias | 0x0102,
    E_NoSetEndOfForkProc    = FSErrorBias | 0x0103,
    E_NoSetBlockSizeProc    = FSErrorBias | 0x0104,
    E_NoReadBlockProc       = FSErrorBias | 0x0105,
    E_NoWriteBlockProc      = FSErrorBias | 0x0106,
    E_NoForkMapBlockProc    = FSErrorBias | 0x0107,
    E_NoReadRangeProc       = FSErrorBias | 0x0108,
    E_NoWriteRangeProc      = FSErrorBias | 0x0109,
};
```

BTree Module Exceptions

These result codes are used internally by the File Manager.

```
enum{
    E_AccessMethodStart     = FSErrorBias | 0x0200,

// BTree Module Errors

    E_BadHeader             = FSErrorBias | 0x0300,
    E_BadRotate             = FSErrorBias | 0x0301,
    E_NotOpenAsBTree        = FSErrorBias | 0x0302, // no BTreeCB allocated for fork
    E_AlreadyOpenAsBTree    = FSErrorBias | 0x0303,
    E_NoBTreeIterator       = FSErrorBias | 0x0308,
    E_BTreeIsEmpty          = FSErrorBias | 0x030A,
    E_NoMoreMapNodes        = FSErrorBias | 0x030B,
    E_BadNodeSize           = FSErrorBias | 0x030C,
    E_BadNodeType           = FSErrorBias | 0x030D,
    E_BadMaxKeyLength       = FSErrorBias | 0x030E,
    E_BadKeyDescriptor      = FSErrorBias | 0x030F,
    E_MinimumKeyTooLong     = FSErrorBias | 0x0310,
    E_RecordWontFit         = FSErrorBias | 0x0311,

    // Existing BTree Errors

    E_BeforeBeginningOfFile  = FSErrorBias | 0x0353,
    E_PastEndOfFile          = FSErrorBias | 0x0354,
    E_UnknownBTreeVersion    = FSErrorBias | 0x0355,
    E_NoKeyCompareProc       = FSErrorBias | 0x0356,
    E_TreeTooDeep            = FSErrorBias | 0x0357,
    E_NoKeyDescriptor        = FSErrorBias | 0x0358,
    E_Reserved               = FSErrorBias | 0x0359,
    E_BadUserID              = FSErrorBias | 0x035A,
    E_UnknownKeyDescType     = FSErrorBias | 0x035B,
    E_BadKeyDescLength       = FSErrorBias | 0x035C,
    E_PlaceMarkerInvalid     = FSErrorBias | 0x035D,
    E_BadKeyField            = FSErrorBias | 0x035E,
    E_BadKeyAttribute        = FSErrorBias | 0x035F,
    E_BadKeyLength           = FSErrorBias | 0x0360,
    E_RecordNotFound         = FSErrorBias | 0x0361,
    E_RecordExists           = FSErrorBias | 0x0362,
    E_NoSpaceLeft            = FSErrorBias | 0x0363,
    E_RecordTooBig           = FSErrorBias | 0x0364,
    E_BadNode                = FSErrorBias | 0x0365,
```

```
    E_NotABTree             = FSErrorBias | 0x0366,
    E_LastBTreeError        = FSErrorBias | 0x03FF,
};
```

Cache Module Exceptions

These result codes are used internally by the File Manager.

```
enum{
    E_NonExist              = FSErrorBias | 0x0400,// cache block does not exist
    E_BufferInUse           = FSErrorBias | 0x0401,// cache buffer is in use
    E_MissingCacheCB        = FSErrorBias | 0x0402,
                                    // cache CB is missing from Fork/Vol CB
    E_CorruptCacheCB        = FSErrorBias | 0x0403,
                                    // cache CB associated with Fork/Vol CB is corrupt
    E_BufferHeaderOverflow  = FSErrorBias | 0x0404,
                                    // cache buffer header is larger than page size
    E_SGListOverflow        = FSErrorBias | 0x0405,
                                    // scatter-gather list's address range is not big
                                    // enough to accomodate the entire block range
    E_InvalidBlockSize      = FSErrorBias | 0x0406,
                            // volume/fork block size is not of the kind 512,1K,2K,4K…
    E_NoPageToRelinquish    = FSErrorBias | 0x0407,
                            // cache release Q is empty; can't relinquish a page to VM
    E_NoEmptyBufferHeaders  = FSErrorBias | 0x0408,
                            // no more empty buffer headers left; time to create more
    E_AlreadyInCache        = FSErrorBias | 0x0409,
                                    // all blocks asked for are already in cache
    E_NoPagesToClean        = FSErrorBias | 0x040A,
                                    // no more pages to clean by page cleaner
};
```

Control Blocks Module Exceptions

These result codes are used internally by the File Manager.

```
enum{
    E_NoCBPtr           = FSErrorBias | 0x0501,// CBPtr is nil
    E_NoSuchPath        = FSErrorBias | 0x0502,
```

```
    E_NoSuchFork        = FSErrorBias | 0x0503,
    E_NoSuchFile        = FSErrorBias | 0x0504,
    E_NoSuchVolume      = FSErrorBias | 0x0505,
    E_NoSuchFSAgent     = FSErrorBias | 0x0506,
    E_InvalidCBType     = FSErrorBias | 0x0507,
    E_InvalidCBPtr      = FSErrorBias | 0x0508,
    E_InvalidParent     = FSErrorBias | 0x0509,
    E_NoFSAgents        = FSErrorBias | 0x050A,
    E_NoQHeadPtr        = FSErrorBias | 0x050B,
    E_ChildStillQueued  = FSErrorBias | 0x050C,
    E_LastCBError       = FSErrorBias | 0x05FF, // Last error for Control Blocks Module
};
```

Object Reference Exceptions

```
enum{
    E_UndefinedObjectRef            = FSErrorBias | 0x0601,
    E_ObjectRefAlreadyInitialized   = FSErrorBias | 0x0602,
    E_ObjectRefAlreadyExists        = FSErrorBias | 0x0603,
    E_InvalidObjectRef              = FSErrorBias | 0x0604,
                                            // kFStheNullObjectRef was passed
};
```

Range Lock Module Exceptions

```
enum{
    E_RangeNotLocked    = FSErrorBias | 0x0701,
                                // tried to unlock a range that wasn't locked
    E_NoAsyncLock       = FSErrorBias | 0x0702,
                                // tried to cancel an async lock, not pending
    E_LockedByOther     = FSErrorBias | 0x0703,
                                // conflicting lock owned by different path
    E_LockedBySelf      = FSErrorBias | 0x0704,
                                // conflicting lock owned by same path
    E_LockedForRead     = FSErrorBias | 0x0705,
                                // conflicting read lock (owner unknown)
};
```

Utilities Module Exceptions

These result codes are used internally by the File Manager.

```
enum{
    E_UtilitiesStart    = FSErrorBias | 0x0800
};
```

Volume Exceptions

```
enum{
    E_MajorOfflineChange    = FSErrorBias | 0x0901,
    E_MinorOfflineChange    = FSErrorBias | 0x0902,
    E_VolumeInUse           = FSErrorBias | 0x0903,
};
```

FSIterator Exceptions

```
enum{
    E_InvalidIterationObjectType    = FSErrorBias | 0x0A01,// can't enter a file object
    E_ExitIteratorScope             = FSErrorBias | 0x0A02,// iterator exited the scope
    E_IteratorScopeException        = FSErrorBias | 0x0A03,
                                        // iterator is undefined due to error or
                                        //  movement of scope locality
    E_UnknownIterationMovement      = FSErrorBias | 0x0A04,
                                        // iterator movement is not defined
    E_InvalidIterationMovement      = FSErrorBias | 0x0A05,
                                        // iterator movement invalid in current context
    E_IteratorOwnership             = FSErrorBias | 0x0A06,// wrong client process ID
    E_EndOfIteration                = FSErrorBias | 0x0A07,
                                        // no objects left to return on iteration
};
```

FSProperty Exceptions

These result codes are used internally by the File Manager.

```
enum{
    E_PropDescrOffsetRangeErr       = FSErrorBias | 0x0B01,
                                        // property descriptor list offset out of range
    E_PropDescrOverrunLengthErr     = FSErrorBias | 0x0B02,
                                        // PropOverrun bitmap is too short
//  reserved                        = FSErrorBias | 0x0B03,
    E_PropDescrNotFoundLengthErr    = FSErrorBias | 0x0B04,
                                        // PropNotFound bit map is too short
    E_PropDescrReadOnlyFailLengthErr = FSErrorBias | 0x0B05,
                                        // PropDescrReadOnlyFail bitmap too short

    E_PropBufferShort               = FSErrorBias | 0x0B06,
                                        // buffer too short for properties.
    E_PropertyBufferFieldTooSmall   = FSErrorBias | 0x0B07,
                                        // from pt of view of supplied buffer field
    // PropertiesGet: field not adequate to hold Property. field is truncated.(Overrun)
    E_PropertyException             = FSErrorBias | 0x0B08,
    E_PropertyBufferFieldTooLarge   = FSErrorBias | 0x0B09,
                                        // from pt of view of supplied buffer field
    //  PropertiesGet: field is larger than needed to hold properties. (Underflow)

    E_CanNotFindDesktopDatabase     = FSErrorBias | 0x0B0A,
                                // either desktopDB or desktopDF not found
    E_CanNotFindPDSFile             = FSErrorBias | 0x0B0B,// cannot find PDS
    E_CanNotFindEDSFile             = FSErrorBias | 0x0B0C,// cannot find EDS
    E_PropertyAlreadyExists         = FSErrorBias | 0x0B0D,
                                // property already exists in desktop database
    E_PropertyTypeError             = FSErrorBias | 0x0B0E,
                                // error getting/setting property in desktop database
    E_PropertyError                 = FSErrorBias | 0x0B0F,
                                // error getting/setting pProperty in desktop database
    E_PropertyReadOnly              = FSErrorBias | 0x0B10,
                                // attempt to set read-only property
};
```

FSDispatch Errors

These result codes are used internally by the File Manager.

```
enum{
    E_UnknownRequest            = FSErrorBias | 0x1001,
                                    // request command ID not defined
    E_VolRecognized             = E_NoError,
                                    // agent was able to recognize the Volume
    E_VolNotRecognized          = FSErrorBias | 0x1002,
                                    // agent was NOT able to recognize the Volume
    E_CanInitialize             = E_NoError,
                                    // agent is able to initialize Volume
    E_CanNotInitialize          = FSErrorBias | 0x1003,
                                    // agent is NOT able to initialize Volume
    E_NotAPath                  = FSErrorBias | 0x1004,// ID used was not for a Path
    E_NotAFork                  = FSErrorBias | 0x1005,// ID used was not for a Fork
    E_NotAFile                  = FSErrorBias | 0x1006,// ID used was not for a File
    E_NotAVolume                = FSErrorBias | 0x1007,// ID used was not for a Volume
    E_NotAnAgent                = FSErrorBias | 0x1008,// ID used was not for an Agent
    E_NotAnObjectIterator       = FSErrorBias | 0x1009,
                                    // ID used was not for an Object Iterator
    E_NotAPropertyIterator      = FSErrorBias | 0x100A,
                                    // ID used was not for a Property Iterator
    E_AgentIncomplete           = FSErrorBias | 0x100B,
                                    // agent did not finish instalation protocol
    E_QueueOverflow             = FSErrorBias | 0x100C,
                                // a notification queue has overflowed the user's size
    E_QueueEmpty                = FSErrorBias | 0x100D,// notification queue is Empty

    // the message length is either larger than the size from the messaging
    // services, or is smaller than implied by the variable data needs

    E_MessageLength             = FSErrorBias | 0x100E,
    E_NoVolumeSpecified         = FSErrorBias | 0x100F,
                                    // no volume was specified for PathnameResolve
                                    // dispatch or request processing task
    E_MessageMustBeAtomic       = FSErrorBias | 0x1010,
    E_MessageMustNotBeAtomic    = FSErrorBias | 0x1011,
    E_MessageSentToWrongObject  = FSErrorBias | 0x1012,

    E_InvalidVolumeSet          = FSErrorBias | 0x1013,
                                // VolumeSetGetInformation: volume set not recognized
```

```
    E_FSShutdown                = FSErrorBias | 0x10F1,
                                    // terminated because system was shutting down
};
```

General File Manager Errors

```
enum{
    E_Unimplemented             = FSErrorBias | 0xFFFF // feature unimplemented
};
```

Glossary

file A collection of data items of any size or content. It is up to each specific file system running under the File Manager to determine the meaning and content of each data item. This more generalized definition of a file allows the File Manager to support such diverse file systems as HFS, UNIX, and DOS FAT.

file system object The basic unit in the Mac OS 8 files environment. Every file, folder, and volume is a file system object, and every file system object is a container for information. For example, volumes can contain folders, files, and properties; folders can contain files, folders, and properties; and files contain properties.

file system object reference A dynamically assigned opaque identifier that is used by almost every File Manager function that refers to an object. Object references are allocated and disposed of on a per-process basis. When you use a File Manager function that returns a file system object reference, the object reference is automatically allocated for your process, and you are responsible for disposing of it.

fork attribute The value attribute of a fork property is the *only* fork attribute; all other attributes of a fork property, such as the fork property's size and type are considered simple. To get or set specific portions of a fork's data, you must use specific stream or memory-mapped file access methods.

object information structure A structure that provides the most commonly used aggregate sets of file, folder, or volume properties. You can get this information all at once, with the `FSObjectGetInformation` and `FSObjectGetVolumeInformation` functions. You cannot set an object information structure as a whole, you can only set individual properties, one at a time, with the `FSObjectSetOneProperty` function.

object iteration The ability to obtain information about one or more file system objects by accessing all available objects that match criteria that you can set. You use object iterators to perform an iteration. See also **object iterator.**

object iterator An object that you use to obtain information about one or more file system objects by accessing all available objects that match your criteria. For example, you can adjust the iterator's movement to go into any embedded containers, and you can make an iterator return files or folders or both types of objects.

property A data item or a set of data that is stored by the file system. Properties can be simple data items, such as dates, file types, and icon definitions; or they can be expandable sets of data such as the data fork and resource fork of a file. Each property has a value, and the value of the property has an actual size and a certain amount of allocated space. A property

cannot exist by itself; it must be contained in a file system object. The properties contained by an object define the object and make it identifiable to the File Manager.

property structure A structure that describes a property of an object. For each property, this structure identifies its creator, selector, attribute, and tag. Each attribute of every property is described by a property structure.

simple attribute Any attributes of any property except the fork property's value attribute. To get or set a simple attribute, you access the data directly, but you get or set all of the value at once. For example, you would get a file's entire creation date as a unit; you couldn't get just the year or month.

simple property The same as a property with a simple attribute. See **simple attribute.**

template constant A constant that provides a more generalized property structure that you can use to get or set any attribute of any property except the value attribute of a fork property.

universe A file system object that represents the user's computer system including all mounted volumes. Because the universe is transient, existing only while the system is running and changing every time the user mounts or unmounts a volume or reconfigures the system in some other way, the properties contained in the universe are transient and are created by the File Manager each time the computer starts up.

value constant A constant that provides a complete property structure that you can use to get or set the value attribute of a simple property.