INSIDE MACINTOSH

Files

Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

ISBN 0-201-63244-6
1 2 3 4 5 6 7 8 9-MU-9695949392
First Printing, August 1992

Contents

Figures, Tables, and Listings

About This Book

This book, *Inside Macintosh: Files,* describes the parts of the Macintosh Operating System that allow you to manage files. It shows in detail how your application can handle the commands typically found in a File menu. It also provides a complete technical reference to the File Manager, the Standard File Package, the Alias Manager, and other file-related services provided by the system software.

If you are new to the Macintosh Operating System, you should begin with the chapter "Introduction to File Management." This chapter describes the basic structure of Macintosh files and the hierarchical file system (HFS) used with Macintosh computers, and it shows how you can use the services provided by the Standard File Package, the File Manager, the Finder, and other system software components to create, open, update, and close files. Because this chapter is designed to be largely self-contained, the reference and summary sections in this chapter are subsets of the corresponding sections from the other chapters in this book.

Once you are familiar with basic file management on Macintosh computers, you might want to read other chapters in this book. The chapter "File Manager" describes how your application can manage shared files; search for specific files in a volume; obtain information about files, directories, and volumes; and perform other advanced operations. This chapter also describes how the File Manager organizes file and directory data on disk and in memory. Much of this information is of interest only to designers of very specialized applications or file-system utility programs.

If you want to customize the user interface for naming and identifying files, you need to read the chapter "Standard File Package." It provides complete information on how to customize and display the dialog boxes that let the user specify the names and locations of files to be saved or opened.

If your application needs to keep track of particular files, directories, or volumes, you might want to use the Alias Manager. It helps you find objects in the file system, even if those objects have been moved or renamed. See the chapter "Alias Manager" for complete details.

The chapter "Disk Initialization Manager" shows how you can initialize disks and erase the contents of previously initialized disks. The Disk Initialization Manager provides a routine that allows you to present the standard user interface for initializing and naming disks. Most applications should call that routine whenever they receive a disk-inserted event and the inserted disk is invalid.

Format of a Typical Chapter

Almost all chapters in this book follow a standard structure. For example, the chapter "Standard File Package" contains these sections:

- "About the Standard File Package." This section provides an overview of the features provided by the Standard File Package.
- "Using the Standard File Package." This section describes the tasks you can accomplish using the Standard File Package. It describes how to use the most common routines, gives related user interface information, provides code samples, and supplies additional information.
- "Standard File Package Reference." This section provides a complete reference to the Standard File Package by describing the data structures and routines that it uses. Each routine description also follows a standard format, which gives the routine declaration and a description of every parameter of the routine. Some routine descriptions also give additional descriptive information, such as assembly-language information or result codes.
- "Summary of the Standard File Package." This section provides the Standard File Package's Pascal interface, as well as the C interface, for the constants, data structures, routines, and result codes associated with the Standard File Package. It also includes relevant assembly-language interface information.

Some chapters contain additional main sections that provide more detailed discussions of certain topics. For example, the chapter "File Manager" contains the section "Identifying Files, Directories, and Volumes," which describes the many ways to identify objects in the file system. That chapter also contains the two advanced sections "Data Organization on Volumes" and "Data Organization in Memory."

Conventions Used in This Book

Inside Macintosh uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain information, such as parameter blocks, use special formats so that you can scan them quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and routines are shown in Courier (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the Glossary.

Types of Notes

There are several types of notes used in this book.

Note

A note like this contains information that is interesting but possibly not essential to an understanding of the main text. (An example appears on page 1-6.) ◆

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. (An example appears on page 1-6.) ▲

▲ **WARNING**

Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. (An example appears on page 1-46.) ▲

Assembly-Language Information

Inside Macintosh provides information about the registers for specific routines like this:

Registers on entry

A0 Contents of register A0 on entry

Registers on exit

D0 Contents of register D0 on exit

In addition, *Inside Macintosh* presents information about the fields of a parameter block in this format:

Parameter block

↔	`inAndOut`	`Integer`	Input/output parameter.
←	`output1`	`Ptr`	Output parameter.
→	`input1`	`Ptr`	Input parameter.

The arrow in the far left column indicates whether the field is an input parameter, output parameter, or both. You must supply values for all input parameters and input/output parameters. The routine returns values in output parameters and input/output parameters.

The second column shows the field name as defined in the MPW Pascal interface files; the third column indicates the Pascal data type of that field. The fourth column provides a brief description of the use of the field. For a complete description of each field, see the discussion that follows the parameter block or the description of the parameter block in the reference section of the chapter.

Development Environment

The system software routines described in this book are available using Pascal, C, or assembly-language interfaces. How you access these routines depends on the development environment you are using. This book shows system software routines in their Pascal interface using the Macintosh Programmer's Workshop (MPW).

All code listings in this book are shown in Pascal. They show methods of using various routines and illustrate techniques for accomplishing particular tasks. All code listings have been compiled and, in most cases, tested. However, Apple Computer does not intend that you use these code samples in your application.

This book occasionally uses *SurfDraw* as the name of a sample application for illustrative purposes; this is not an actual product of Apple Computer, Inc.

APDA, Apple's source for developer tools, offers worldwide access to a broad range of programming products, resources, and information for anyone developing on Apple platforms. You'll find the most current versions of Apple and third-party development tools, debuggers, compilers, languages, and technical references for all Apple platforms. To establish an APDA account, obtain additional ordering information, or find out about site licensing and developer training programs, contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, M/S 33-G
Cupertino, CA 95014-6299

Telephone:	800-282-2732 (United States) 800-637-0029 (Canada) 800-562-3910 (elsewhere in the world)
Fax:	408-562-3971
Telex:	171-576

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information on registering signatures, file types, Apple events, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 75-3T
Cupertino, CA 95014-6299

Introduction to File Management

Contents

Introduction to File Management

This chapter is a general introduction to file management on Macintosh computers. It explains the basic structure of Macintosh files and the hierarchical file system (HFS) used with Macintosh computers, and it shows how you can use the services provided by the Standard File Package, the File Manager, the Finder, and other system software components to create, open, update, and close files.

You should read this chapter if your application implements the commands typically found in an application's File menu—except for printing commands and the Quit command, which are described elsewhere. This chapter describes how to

- create a new file
- open an existing file
- close a file
- save a document's data in a file
- save a document's data in a file under a new name
- revert to the last saved version of a file
- create and read a preferences file

Depending on the requirements of your application, you may be able to accomplish all your file-related operations by following the instructions given in this chapter. If your application has more specialized file management needs, you'll need to read some or all of the remaining chapters in this book.

This chapter assumes that your application is running in an environment in which the routines that accept file system specification records (defined by the FSSpec data type) are available. File system specification records, introduced in system software version 7.0, simplify the identification of objects in the file system. Your development environment may provide "glue" that allows you to call those routines in earlier system software versions. If such glue is not available and you want your application to run in system software versions earlier than version 7.0, you need to read the discussion of HFS file-manipulation routines in the chapter "File Manager" in this book.

This chapter begins with a description of files and their organization into directories and volumes. Then it describes how to test for the presence of the routines that accept FSSpec records and how to use those routines to perform the file management tasks listed above. The chapter ends with descriptions of the data structures and routines used to perform these tasks. The "File Management Reference" and "Summary of File Management" sections in this chapter are subsets of the corresponding sections of the remaining chapters in this book.

About Files

To the user, a file is simply some data stored on a disk. To your application, a **file** is a named, ordered sequence of bytes stored on a Macintosh volume, divided into two forks (as described in the following section, “File Forks”). The information in a file can be used for any of a variety of purposes. For example, a file might contain the text of a letter or the numerical data in a spreadsheet; these types of files are usually known as documents. Typically a **document** is a file that a user can create and edit. A document is usually associated with a single application, which the user expects to be able to open by double-clicking the document’s icon in the Finder.

A file might also contain an application. In that case, the information in the file consists of the executable code of the application itself and any application-specific resources and data. Applications typically allow the user to create and manipulate documents. Some applications also create special files in which they store user-specific settings; such files are known as **preferences files.**

The Macintosh Operating System also uses files for other purposes. For example, the File Manager uses a special file located in a volume to maintain the hierarchical organization of files and folders in that volume. This special file is called the volume’s **catalog file.** Similarly, if virtual memory is in operation, the Operating System stores unused pages of memory in a disk file called the **backing-store file.**

No matter what its function, each file shares certain characteristics with every other file. This section describes these general characteristics of Macintosh files, including

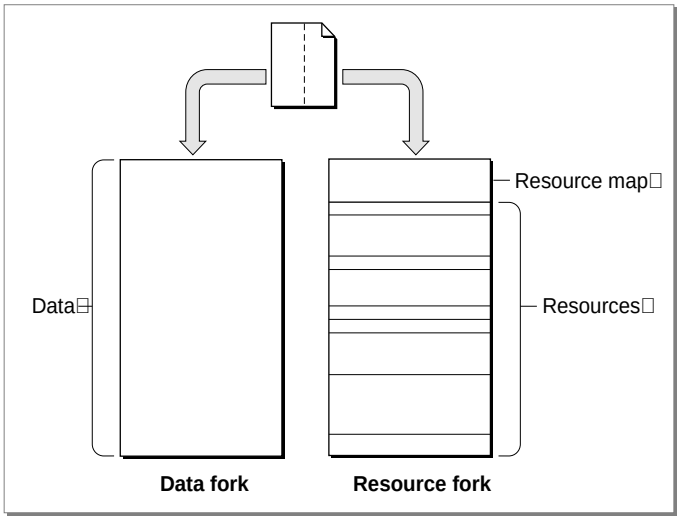
- file forks
- file size and access characteristics
- file system organization
- file naming and identification

File Forks

Many operating systems treat a file simply as a named, ordered sequence of bytes (possibly terminated by a byte having a special value that indicates the end-of-file). As illustrated in Figure 1-1, however, each Macintosh file has two **forks,** known as the data fork and the resource fork.

A file’s **resource fork** contains that file’s resources. If the file is an application, the resource fork typically contains resources that describe the application’s menus, dialog boxes, icons, and even the executable code of the application itself. A particularly important resource is the application’s `'SIZE'` resource, which contains information about the capabilities of the application and its run-time memory requirements. If the file is a document, its resource fork typically contains preference settings, window locations, and document-specific fonts, icons, and so forth.

Figure 1-1 The two forks of a Macintosh file



A file's **data fork** contains the file's data. It is simply a series of consecutive bytes of data. In a sense, the data fork of a Macintosh file corresponds to an entire file in operating systems that treat a file simply as a sequence of bytes. The bytes stored in a file's data fork do not have to exhibit any internal structure, unlike the bytes stored in the resource fork (which consists of a resource map followed by resources). Rather, your application is responsible for interpreting the bytes in the data fork in whatever manner is appropriate. The data fork of a document file might, for example, contain the text of a letter.

Even though a Macintosh file always contains both a resource fork and a data fork, one or both of those forks can be empty. Document files sometimes contain only data (in which case the resource fork is empty). More often, document files contain both resources and data. Application files generally contain resources only (in which case, the data fork is empty). Application files can, however, contain data as well.

Whether you store specific data in the data fork or in the resource fork of a file depends largely on whether that data can usefully be structured as a resource. For example, if you want to store a small number of names and telephone numbers, you can easily define a resource type that pairs each name with its telephone number. Then you can read names and corresponding numbers from the resource file by using Resource Manager routines. To retrieve the data stored in a resource, you simply specify the resource type and ID; you don't need to know, for instance, how many bytes of data are stored in that resource.

In some cases, however, it is not possible or advisable to store your data in resources. The data might be too difficult to put into the structure required by the Resource Manager. For example, it is easiest to store a document's text, which is usually of variable length, in a file's data fork. Then you can use File Manager routines to access any byte or group of bytes individually.

Even when it is easy to define a resource type for your data, limitations on the Resource Manager might compel you to store your data in the data fork instead. A resource fork can contain at most about 2700 resources. More importantly, the Resource Manager searches linearly through a file's resource types and resource IDs. If the number of types or IDs to be searched is large, accessing the resource data can become slow. As a rule of thumb, if you need to manage data that would occupy more than about 500 resources total, you should use the data fork instead.

IMPORTANT

In general, you should store data created by the user in a file's data fork, unless the data is guaranteed to occupy a small number of resources. The Resource Manager was not designed to be a general-purpose data storage and retrieval system. Also, the Resource Manager does not support multiple access to a file's resource fork. If you want to store data that can be accessed by multiple users of a shared volume, use the data fork. ▲

Because the Resource Manager is of limited use in storing large amounts of user-generated data, most of the techniques in "Using Files" (beginning on page 1-12) illustrate the use of File Manager routines to manage information stored in a file's data fork. See the section "Using a Preferences File" on page 1-36 for an example of the use of the Resource Manager to access data stored in a file's resource fork.

File Size

The size of a file is usually limited only by the size of its volume. A **volume** is a portion of a storage device that is formatted to contain files. A volume can be an entire disk or only a part of a disk. A 3.5-inch floppy disk, for instance, is always formatted as one volume. Other memory devices, such as hard disks and file servers, can contain multiple volumes.

Note

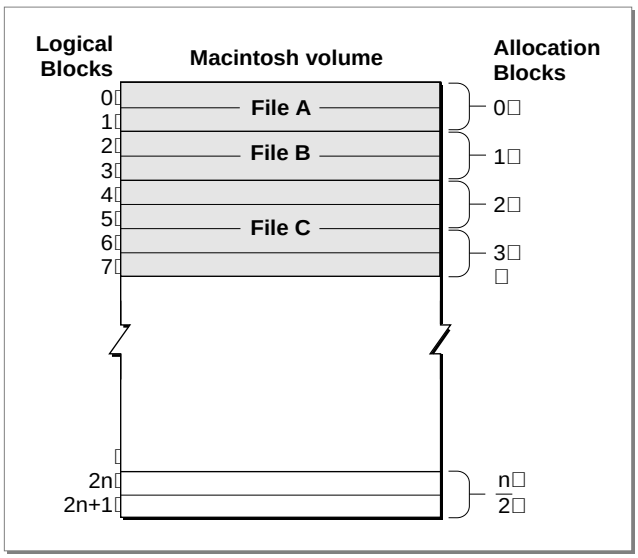
Actually, a file on an HFS volume can be as large as 2 GB ($7FFFFFFF bytes). Most volumes are not large enough to hold a file of that size. An HFS volume currently can be as large as 2 GB. ◆

The size of a volume varies from one type of device to another. Volumes are formatted into chunks known as **logical blocks,** each of which can contain up to 512 bytes. A double-sided 3.5-inch floppy disk, for instance, usually has 1600 logical blocks, or 800 KB.

Generally, however, the size of a logical block on a volume is of interest only to the disk device driver. This is because the File Manager always allocates space to a file in units called allocation blocks. An **allocation block** is a group of consecutive logical blocks. The File Manager can access a maximum of 65,535 allocation blocks on any volume. For small volumes, such as volumes on floppy disks, the File Manager uses an allocation block size of one logical block. To support volumes larger than about 32 MB, the File

Manager needs to use an allocation block size that is at least two logical blocks. To support volumes larger than about 64 MB, the File Manager needs to use an allocation block that is at least three allocation blocks. In this way, by progressively increasing the number of logical blocks in an allocation block, the File Manager can handle larger and larger volumes. Figure 1-2 illustrates how logical blocks are grouped into allocation blocks.

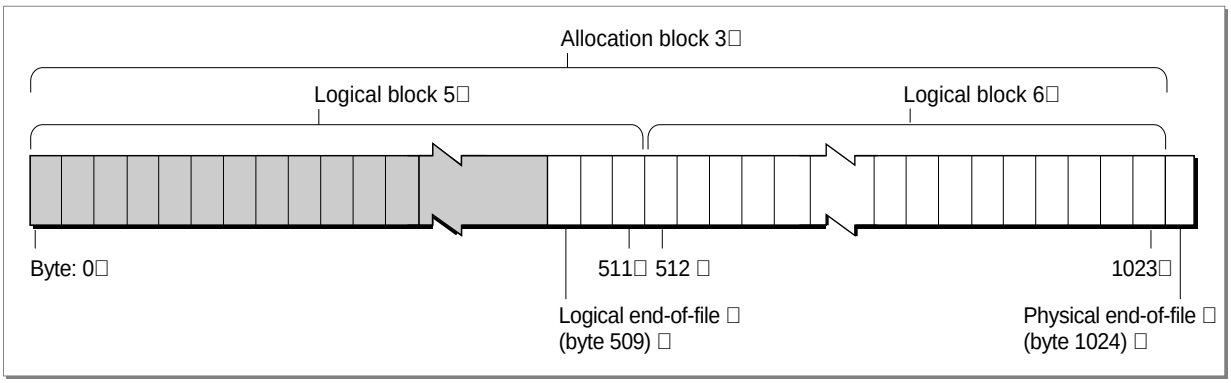
Figure 1-2 Logical blocks and allocation blocks



The size of the allocation blocks on a volume is determined when the volume is initialized and depends on the number of logical blocks it contains. In general, the Disk Initialization Manager uses the smallest allocation block size that will allow the File Manager to address the entire volume. A nonempty file fork always occupies at least one allocation block, no matter how many bytes of data that file fork contains. On a 40 MB volume, for example, a file's data fork occupies at least 1024 bytes (that is, two logical blocks), even if it contains only 11 bytes of actual data.

To distinguish between the amount of space allocated to a file and the number of bytes of actual data in the file, two numbers are used to describe the size of a file. The **physical end-of-file** is the number of bytes currently allocated to the file; it's 1 greater than the number of the last byte in its last allocation block (since the first byte is byte number 0). As a result, the physical end-of-file is always an exact multiple of the allocation block size. The **logical end-of-file** is the number of those allocated bytes that currently contain data; it's 1 greater than the number of the last byte in the file that contains data. For example, on a volume having an allocation block size of two logical blocks (that is, 1024 bytes), a file with 509 bytes of data has a logical end-of-file of 509 and a physical end-of-file of 1024 (see Figure 1-3).

Figure 1-3 Logical end-of-file and physical end-of-file



You can move the logical end-of-file to adjust the size of the file. When you move the logical end-of-file to a position more than one allocation block short of the current physical end-of-file, the File Manager automatically deletes the unneeded allocation block from the file. Similarly, you can increase the size of a file by moving the logical end-of-file past the physical end-of-file. When you move the logical end-of-file past the physical end-of-file, the File Manager automatically adds one or more allocation blocks to the file. The number of allocation blocks added to the file is determined by the volume's clump size. A **clump** is a group of contiguous allocation blocks. The purpose of enlarging files always by adding clumps is to reduce file fragmentation on a volume, thus improving the efficiency of read and write operations.

If you plan to keep extending a file with multiple write operations and you know in advance approximately how large the file is likely to become, you should first call the `SetEOF` function to set the file to that size (instead of having the File Manager adjust the size each time you write past the end-of-file). Doing this reduces file fragmentation and improves I/O performance.

File Access Characteristics

A file can be open or closed. Your application can perform certain operations, such as reading and writing data, only on open files. It can perform other operations, such as deleting, only on closed files.

When you open a file, the File Manager reads information about the file from its volume and stores that information in a **file control block** (FCB). The File Manager also creates an **access path** to the file, a description of the route to be followed when accessing the file. The access path specifies the volume on which the file is located and the location of the file on the volume. Each access path is assigned a unique **file reference number** (some number greater than 0) by which your application refers to the path. Multiple access paths can be opened to the same file.

For each open access path to a file, the File Manager maintains a current position marker, called the **file mark,** to keep track of where it is in the file during a read or write operation. The mark is the number of the next byte that will be read or written; each time a byte is read or written, the mark is moved. When, during a write operation, the mark reaches the number of the last byte currently allocated to the file, the File Manager adds another clump to the file.

You can read bytes from and write bytes to a file either singly or in sequences of virtually unlimited length. You can specify where each read or write operation should begin by setting the mark or specifying an offset; if you don't, the operation begins at the current file mark.

Each time you want to read or write a file's data, you need to pass the address of a **data buffer,** a part of RAM (usually in your application's heap). The File Manager uses the buffer when it transfers data to or from your application. You can use a single buffer for each read or write operation, or change the address and size of the buffer as necessary.

When your application writes data to a file, the File Manager transfers the data from your application's data buffer and writes it to the **disk cache,** a part of RAM (usually in the System heap). The File Manager uses the disk cache as an intermediate buffer when reading data from or writing it to the file system. When your application requests that data be read from a file, the File Manager looks for the data in the disk cache and transfers it to your application's data buffer if the data is found in the cache; otherwise, the File Manager reads the requested bytes from the disk and puts them in your data buffer.

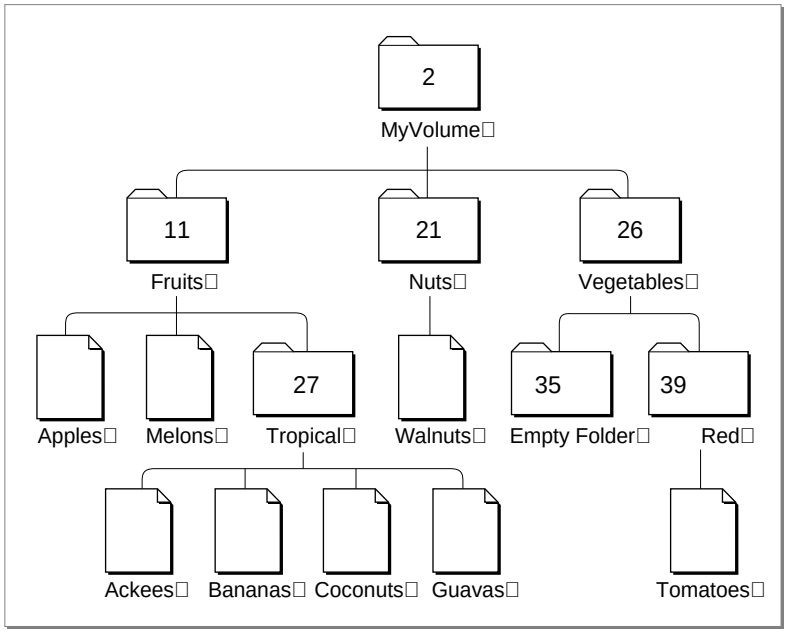
Note

You can also read a continuous stream of characters or a line of characters from a file. In the first case, you ask the File Manager to read a specific number of bytes: When that many have been read, or when the mark reaches the logical end-of-file, the read operation terminates. In the second case, called **newline mode**, the read operation terminates when either of the above conditions is met or when a specified character, the newline character, is read. The **newline character** is usually Return (ASCII code $0D), but it can be any character. Information about newline mode is associated with each access path to a file and can differ from one access path to another. See the chapter "File Manager" in this book for more information about newline mode. ◆

The Hierarchical File System

The Macintosh Operating System uses a method of organizing files called the **hierarchical file system (HFS).** In HFS, files are grouped into **directories** (also called **folders**), which themselves are grouped into other directories, as illustrated in Figure 1-4. The number listed for each directory is its **directory ID.** The directory ID is one component of a file system specification, as explained in the next section, "Identifying Files and Directories."

Figure 1-4 The Macintosh hierarchical file system



The Finder is responsible for managing the files and folders on the desktop. It works with the File Manager to maintain the organization of files and folders on a volume. The hierarchical relationship of folders within folders on the desktop corresponds directly to the hierarchical directory structure maintained on the volume. The volume is known as the **root directory,** and the folders are known as subdirectories, or simply directories.

A volume appears on the desktop only after it has been mounted. Ejectable volumes (such as 3.5-inch floppy disks) are mounted when they're inserted into a disk drive; nonejectable volumes (such as those on hard disks) are mounted automatically at system startup. When a volume is **mounted,** the File Manager places information about the volume in a nonrelocatable block of memory called a **volume control block (VCB).** The number of volumes that can be mounted at any time is limited only by the number of drives attached and available memory.

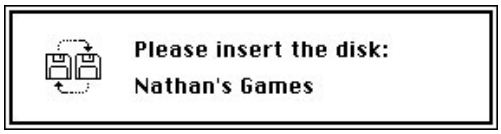
When a volume is mounted, the File Manager assigns a **volume reference number** by which you can refer to the volume for as long as it remains mounted. You can also identify a volume by its **volume name,** a sequence of 1 to 27 printing characters, excluding colons (:). (The File Manager ignores case when comparing names but does recognize diacritical marks.) Whenever possible, though, you should use the volume reference number to avoid confusion between volumes with the same name.

Note

A volume reference number is valid only until the volume is unmounted. If a single volume is mounted and then unmounted, the File Manager may assign it a different volume reference number when it is next mounted. ◆

When an application ejects a 3.5-inch disk from a drive, the File Manager places the volume **offline.** When a volume is offline, the volume control block is kept in memory and the volume reference number is still valid. If you make a File Manager call that specifies that volume, the File Manager presents the disk switch dialog box to the user. Figure 1-5 shows a sample disk switch dialog box.

Figure 1-5 The disk switch dialog box



When the user drags a volume icon to the Trash, that volume is **unmounted**; the volume control block is released, and the volume is no longer known to the File Manager. In particular, the volume reference number previously assigned to the volume is no longer valid.

Each subdirectory is located within a directory called its **parent directory.** Typically, the parent directory is specified by a **parent directory ID,** which is simply the directory ID of the parent directory. The File Manager assigns a special parent directory ID to a volume's root directory. This is primarily to permit a consistent method of identifying files and directories using the volume reference number, the parent directory ID, and the file or directory name. See the next section, "Identifying Files and Directories," for details.

For the most part, your application does not need to be concerned about, or keep track of, the location of files in the file system hierarchy. Most of the files your application opens and saves are specified by the user or another application, and their location is provided to your application by either the Finder or the Standard File Package. One notable exception here concerns preferences files, which are typically stored in the Preferences folder in the currently active System Folder. See "Using a Preferences File" on page 1-36 for instructions on finding preferences files.

Note

In addition to files, folders, and volumes, a fourth type of object, namely an alias, might appear on the Finder desktop. An **alias** is a special kind of file that represents another file, folder, or volume. The Finder and the Standard File Package automatically resolve aliases before passing files to your application, so you generally don't need to do anything with aliases. For more information on working with alias files, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* and the chapter "Alias Manager" in this book. ◆

Identifying Files and Directories

The hierarchical arrangement of files and directories allows you to identify a file or directory uniquely by providing just three pieces of information: its volume reference number, its parent directory ID, and its name within that parent directory. The system software lets you specify these three items together in a file system specification record, defined by the `FSSpec` data type:

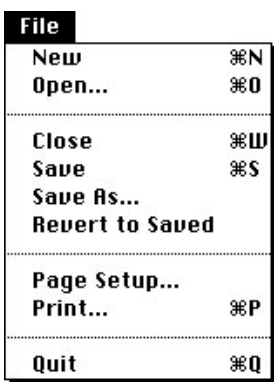
```
TYPE FSSpec    =           {file system specification}
RECORD
   vRefNum:    Integer;    {volume reference number}
   parID:      LongInt;    {directory ID of parent directory}
   name:       Str63;      {filename or directory name}
END;
```

The `FSSpec` record provides a simple and standard format for specifying files and directories. For example, the Standard File Package procedure `StandardGetFile` uses an `FSSpec` record to return information identifying a user-selected file or folder. You can pass that specification directly to any file-manipulation routines, such as `FSpOpenDF` and `FSpDelete`, that accept `FSSpec` records. In addition, the Alias Manager, Edition Manager, and Finder all use `FSSpec` records to specify files and directories.

Using Files

This section describes how to perform typical file operations using some of the services provided by the Standard File Package, the File Manager, the Finder, and other system software components. Figure 1-6 shows the typical appearance of an application's File menu.

Figure 1-6 A typical File menu



Note that all the commands in this menu, except for the Quit and Page Setup commands, manipulate files. Your application's File menu should resemble the menu shown in Figure 1-6 as closely as possible. In general, whenever the user creates or manipulates information that is stored in a document, you need to implement all the commands shown in Figure 1-6.

Note

Some applications allow the user to create or edit information that is not stored in a document. In those cases, it is inappropriate to put the commands that create or manipulate that information in the File menu. Instead, group those commands together in a separate menu. ◆

Listing 1-1 shows one way to handle some of the typical commands in a File menu. Most of the techniques described in this section are illustrated by means of definitions of the functions called in Listing 1-1.

Listing 1-1 Handling the File menu commands

```
PROCEDURE DoHandleFileCommand (menuItem: Integer);
VAR
   myErr: OSErr;
BEGIN
   CASE menuItem OF
      iNew:
         myErr := DoNewCmd;      {create a new document}
      iOpen:
         myErr := DoOpenCmd;     {open an existing document}
      iClose:
         myErr := DoCloseCmd;    {close the current document}
      iSave:
         myErr := DoSaveCmd;     {save the current document}
      iSaveAs:
         myErr := DoSaveAsCmd;   {save document under new name}
      iRevert:
         myErr := DoRevertCmd;   {revert to last saved version}
      OTHERWISE
         ;
   END;
END;
```

Your application should deactivate any menu commands that do not apply to the frontmost window. For example, if the frontmost window is not a document window belonging to your application, then the Close, Save, Save As, and Revert commands should be dimmed when the menu appears. Similarly, if the document in the frontmost window does belong to your application but contains data that has not changed since it

was last saved, then the Save menu command should be dimmed. See "Adjusting the File Menu" on page 1-37 for details on implementing this feature. The definitions of the application-defined functions used in Listing 1-1 assume that this feature has been implemented.

The techniques described in this chapter for manipulating files assume that you identify files and directories by using file system specification records. Because the routines that accept FSSpec records are not available on all versions of system software, you may need to test for the availability of those routines before actually calling any of them. See the next section, "Testing for File Management Routines," for details.

Testing for File Management Routines

To determine the availability of the routines that operate on FSSpec records, you can call the Gestalt function with the gestaltFSAttr selector code, as illustrated in Listing 1-2.

Listing 1-2 Testing for the availability of routines that operate on FSSpec records

```
FUNCTION FSSpecRoutinesAvail: Boolean;
VAR
   myErr:         OSErr;      {Gestalt result code}
   myFeature:     LongInt;    {Gestalt response}
BEGIN
   FSSpecRoutinesAvail := FALSE;
   IF gHasGestalt THEN        {if Gestalt is available}
      BEGIN
         myErr := Gestalt(gestaltFSAttr, myFeature);
         IF myErr = noErr THEN
            IF BTst(myFeature, gestaltHasFSSpecCalls) THEN
               FSSpecRoutinesAvail := TRUE;
      END;
END;
```

To use the procedures defined in the following sections to open and save files, you also need to make sure that the routines StandardGetFile and StandardPutFile are available. You can do this by passing Gestalt the gestaltStandardFileAttr selector and verifying that the bit gestaltStandardFile58 is set in the response value. Also, before using the FindFolder function (as shown, for example, in Listing 1-10 on page 1-25), you should call the Gestalt function with the gestaltFindFolderAttr selector and verify that the gestaltFindFolderPresent bit is set; this indicates that the FindFolder function is available.

If the routines that operate on FSSpec records are not available, you can use corresponding File Manager and Standard File Package routines. For example, if you cannot call FSpOpenDF, you can call HOpenDF. That is, instead of writing

```
myErr := FSpOpenDF(mySpec, fsCurPerm, myFile);
```

you can write something like

```
myErr := HOpenDF(myVol, myDirID, myName, fsCurPerm, myFile);
```

The only difference is that the mySpec parameter is replaced by three parameters specifying the volume reference number, the parent directory ID, and the filename. With only a few exceptions, all of the techniques presented in this chapter can be easily adapted to work with high-level HFS routines in place of the routines that work with FSSpec records.

Note
One notable exception concerns the Standard File Package procedures SFGetFile and SFPutFile. The vRefNum field of the reply record passed to both these functions contains a **working directory reference number,** which encodes both the directory ID and the volume reference number. In general, you should avoid using this number; instead you can turn it into the corresponding directory ID and volume reference number by calling the GetWDInfo function. See the chapter "File Manager" in this book for details on working directory reference numbers. ◆

Defining a Document Record

When a user creates a new document or opens an existing document, your application displays the contents of the document in a window, which provides a standard interface for the user to view and possibly edit the document data. It is useful for your application to define a **document record,** an application-specific data structure that contains information about the window, any controls in the window (such as scroll bars), and the file (if any) whose contents are displayed in the window. Listing 1-3 illustrates a sample document record for an application that handles text files.

Listing 1-3 A sample document record

```
TYPE
   MyDocRecHnd    = ^MyDocRecPtr;
   MyDocRecPtr    = ^MyDocRec;
   MyDocRec       =
   RECORD
      editRec:       TEHandle;         {handle to TextEdit record}
      vScrollBar:    ControlHandle;    {vertical scroll bar}
```

```
      hScrollBar:    ControlHandle;    {horizontal scroll bar}
      fileRefNum:    Integer;          {ref num for window's file}
      fileFSSpec:    FSSpec;           {file's FSSpec}
      windowDirty:   Boolean;          {has window data changed?}
   END;
```

Some fields in the `MyDocRec` record hold information about the TextEdit record that contains the window's text data. Other fields describe the horizontal and vertical scroll bars in the window. The `myDocRec` record also contains a field for the file reference number of the open file (if any) whose data is displayed in the window and a field for the file system specification that identifies that file. The file reference number is needed when the application manipulates the open file (for example, when it reads data from or writes data to the file, and when it closes the file). The `FSSpec` record is needed when a "safe-save" procedure is used to save data in a file.

The last field of the `MyDocRec` data type is a Boolean value that indicates whether the contents of the document in the TextEdit record differ from the contents of the document in the associated file. When your application first reads a file into the window, you should set this field to `FALSE`. Then, when any subsequent operations alter the contents of the document, you should set the field to `TRUE`. Your application can inspect this field whenever appropriate to determine if special processing is needed. For example, when the user closes a document window and the value of the `windowDirty` flag is `TRUE`, your application should ask the user whether to save the changed version of the document in the file. See Listing 1-16 (page 1-33) for details.

To associate a document record with a particular window, you can simply set a handle to that record as the reference constant of the window (by using the Window Manager procedure `SetWRefCon`). Then you can retrieve the document record by calling the `GetWRefCon` function. Listing 1-15 illustrates this process.

Creating a New File

The user expects to be able to create a new document using the New command in the File menu. Listing 1-4 illustrates one way to handle the New menu command.

Listing 1-4 Handling the New menu command

```
FUNCTION DoNewCmd: OSErr;
VAR
   myWindow:   WindowPtr;  {the new document window; ignored here}
BEGIN
   {Create a new window and make it visible.}
   DoNewCmd := DoNewDocWindow(TRUE, myWindow);
END;
```

The DoNewCmd function simply calls the application-defined function DoNewDocWindow (shown in Listing 1-6). The first parameter to DoNewDocWindow determines whether the new window should be visible or not; the value TRUE indicates that the new window should be visible. If DoNewDocWindow completes successfully, it returns a window pointer to the calling routine in the second parameter. The DoNewCmd function ignores that returned window pointer.

Listing 1-5 Creating a new document window

```
FUNCTION DoNewDocWindow (newDocument: Boolean; var myWindow: WindowPtr):
                                    OSErr;
VAR
   myData:     MyDocRecHnd;   {the window's data record}
CONST
   rDocWindow = 1000;         {resource ID of window template}
BEGIN
   {Allocate a new window; see Window Mgr chapter for details.}
   myWindow := GetNewWindow(rDocWindow, NIL, WindowPtr(-1));
   IF myWindow = NIL THEN
      BEGIN
         DoNewDocWindow := MemError;
         Exit(DoNewDocWindow);
      END;

   {Allocate space for the window's data record.}
   myData := MyDocRecHnd(NewHandle(SizeOf(MyDocRec)));
   IF myData = NIL THEN
      BEGIN
         DoNewDocWindow := MemError;
         DisposeWindow(myWindow);
         Exit(DoNewDocWindow);
      END;

   MoveHHi(Handle(myData));               {move the handle high}
   HLock(Handle(myData));                 {lock the handle}
   WITH myData^^ DO                       {fill in window data}
      BEGIN
         editRec := TENew(gDestRect, gViewRect);
         vScroll := GetNewControl(rVScroll, myWindow);
         hScroll := GetNewControl(rHScroll, myWindow);
         fileRefNum := 0;                 {no file yet!}
         windowDirty := FALSE;
         IF (editRec = NIL) OR (vScroll = NIL) OR (hScroll = NIL) THEN
```

```
            BEGIN
               DoNewDocWindow := memFullErr;
               DisposeWindow(myWindow);
               DisposeControl(vScroll);
               DisposeControl(hScroll);
               TEDispose(editRec);
               DisposeHandle(myData);
               Exit(DoNewDocWindow);
            END;
      END;

   IF newDocument THEN                    {if new document, show it}
      ShowWindow(myWindow);

   SetWRefCon(myWindow, LongInt(myData)); {link record to window}
   HUnlock(Handle(myData));               {unlock the handle}
   DoNewDocWindow := noErr;
END;
```

Note that the `DoNewDocWindow` function does not actually create a new file. The reason for this is that it is usually better to wait until the user actually saves a new document before creating a file (mainly because the user might decide not to save the document). The `DoNewDocWindow` function creates a window, allocates a new document record, and fills out the fields of that record. However, it sets the `fileRefNum` field of the document record to 0 to indicate that no file is currently associated with this window.

Opening a File

Your application might need to open a file in several different situations. For example, if the user launches your application by double-clicking one of its document icons in the Finder, the Finder provides your application with information about the selected file (if your application receives high-level events, the Finder sends it an Open Documents event). At that point, you want to create a new window for the document and read the document data from the file into the window.

Your application also opens files after the user chooses the Open command in the File menu. In this case, you need to determine which file to open. You can use the Standard File Package to present a standard dialog box that allows the user to navigate the file system hierarchy (if necessary) and select a file of the appropriate type. Once you get the necessary information from the Standard File Package, you can then create a new window for the document and read the document data from the file into the window.

As you can see, it makes sense to divide the process of opening a document into several different routines. You can have a routine that elicits a file selection from the user and another routine that creates a window and reads the file data into it. In the sample

listings given here, the function `DoOpenCmd` handles the interaction with the user and `DoOpenFile` reads a file into a new window.

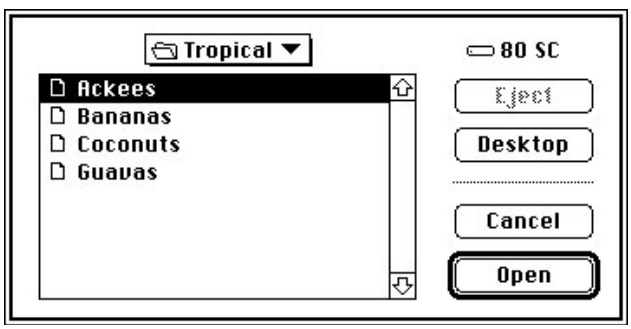
Listing 1-6 shows one way to handle the Open command in the File menu. It uses the Standard File Package routine `StandardGetFile` to determine which file the user wants to open.

Listing 1-6 Handling the Open menu command

```
FUNCTION DoOpenCmd: OSErr;
VAR
   myReply:    StandardFileReply;   {Standard File reply record}
   myTypes:    SFTypeList;          {types of files to display}
   myErr:      OSErr;
BEGIN
   myErr := noErr;
   myTypes[0] := 'TEXT';            {display text files only}
   StandardGetFile(NIL, 1, myTypes, myReply);
   IF myReply.sfGood THEN
      myErr := DoOpenFile(myReply.sfFile)
   ELSE
      myErr := usrCanceledErr;
   DoOpenCmd := myErr;
END;
```

The `StandardGetFile` procedure requires a list of file types to display in an Open dialog box, as in Figure 1-7. In this case, only text files are to be listed.

Figure 1-7 The default Open dialog box



The user can scroll through the list of files in the current directory, change the current directory, select a file to open, or cancel the operation altogether. When the user clicks either the Cancel or the Open button, `StandardGetFile` fills out the Standard File reply record you pass to it, which has this structure:

```
TYPE StandardFileReply =
   RECORD
      sfGood:        Boolean;    {TRUE if user did not cancel}
      sfReplacing:   Boolean;    {TRUE if replacing file with same name}
      sfType:        OSType;     {file type}
      sfFile:        FSSpec;     {selected item}
      sfScript:      ScriptCode; {script of selected item's name}
      sfFlags:       Integer;    {Finder flags of selected item}
      sfIsFolder:    Boolean;    {selected item is a folder}
      sfIsVolume:    Boolean;    {selected item is a volume}
      sfReserved1:   LongInt;    {reserved}
      sfReserved2:   Integer;    {reserved}
   END;
```

In this situation, the relevant fields of the reply record are the `sfGood` and `sfFile` fields. If the user selects a file to open, the `sfGood` field is set to `TRUE` and the `sfFile` field contains an `FSSpec` record for the selected file. In Listing 1-6, the returned `FSSpec` record is passed directly to the application-defined function `DoOpenFile`. Listing 1-7 illustrates a way to define the `DoOpenFile` function.

Listing 1-7 Opening a file

```
FUNCTION DoOpenFile (mySpec: FSSpec): OSErr;
VAR
   myWindow:      WindowPtr;          {window for file data}
   myData:        MyDocRecHnd;        {handle to window data}
   myFileRefNum:  Integer;            {file reference number}
   myErr:         OSErr;
BEGIN
   {Create a new window, but don't show it yet.}
   myErr := DoNewDocWindow(FALSE, myWindow);
   IF (myErr <> noErr) OR (myWindow = NIL) THEN
      BEGIN
         DoOpenFile := myErr;
         Exit(DoOpenFile);
      END;

   SetWTitle(myWindow, mySpec.name);   {set window's title}
   MySetWindowPosition(myWindow);      {set window position}
```

```
   {Open the file's data fork for reading and writing.}
   myErr := FSpOpenDF(mySpec, fsRdWrPerm, myFileRefNum);
   IF myErr <> noErr THEN
      BEGIN
         DisposeWindow(myWindow);
         DoOpenFile := myErr;
         Exit(DoOpenFile);
      END;

   {Retrieve handle to window's data record.}
   myData := MyDocRecHnd(GetWRefCon(myWindow));
   myData^^.fileRefNum := myFileRefNum;{save file information}
   myData^^.fileFSSpec := mySpec;

   myErr := DoReadFile(myWindow);      {read in file data}
   ShowWindow(myWindow);               {now show the window}
   DoOpenFile := myErr;
END;
```

This function is relatively simple because much of the real work is done by the two functions `DoNewDocWindow` and `DoReadFile`. The `DoReadFile` function is responsible for actually reading the file data from the disk into the TextEdit record associated with the document window. See the next section, “Reading File Data,” for a sample definition of `DoReadFile`.

In Listing 1-7, the key step is the call to `FSpOpenDF`, which opens the data fork of the specified file. A file reference number—which indicates an **access path** to the open file—is returned in the third parameter. As you can see, this reference number is saved in the document record, from where it can easily be retrieved for future calls to the `FSRead` and `FSWrite` functions.

The second parameter in a call to the `FSpOpenDF` function specifies the **access mode** for opening the file. For each file, the File Manager maintains access mode information that determines what type of access is available. Most applications support one of two types of access:

- A single user is allowed to read from and write to a file.
- Multiple users are allowed to read from a file, but no one can write to it.

Your application can use the following constants to specify these types of access:

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

To open a file with exclusive read/write access, you can specify `fsRdWrPerm`. To open a file with read-only access, specify `fsRdPerm`. If you want to open a file and don't know or care which type of access is available, specify `fsCurPerm`. When you specify `fsCurPerm`, if no access paths are already open, the file is opened with exclusive read/write access. If other access paths are already open, but they are read-only, another read-only path is opened.

Reading File Data

Once you have opened a file, you can read data from it by calling the `FSRead` function. Generally you need to read data from a file when the user first opens a file or when the user reverts to the last saved version of a document. The `DoReadFile` function defined in Listing 1-8 illustrates how to use `FSRead` to read data from a file into a TextEdit record in either situation.

Listing 1-8 Reading data from a file

```
FUNCTION DoReadFile (myWindow: WindowPtr): OSErr;
VAR
   myData:     MyDocRecHnd;           {handle to a document record}
   myFile:     Integer;               {file reference number}
   myLength:   LongInt;               {number of bytes to read from file}
   myText:     TEHandle;              {handle to TextEdit record}
   myBuffer:   Ptr;                   {pointer to data buffer}
   myErr:      OSErr;
BEGIN
   myData := MyDocRecHnd(GetWRefCon(myWindow)); {get window's data}
   myFile := myData^^.fileRefNum;               {get file reference number}
   myErr := SetFPos(myFile, fsFromStart, 0);    {set file mark at start}
   IF myErr <> noErr THEN
      BEGIN
         DoReadFile := myErr;
         Exit(DoReadFile);
      END;

   myErr := GetEOF(myFile, myLength);           {get file length}
   myBuffer := NewPtr(myLength);                {allocate a buffer}
   IF myBuffer = NIL THEN
      BEGIN
         DoReadFile := MemError;
         Exit(DoReadFile);
      END;
```

```
   myErr := FSRead(myFile, myLength, myBuffer); {read data into buffer}
   IF (myErr = noErr) OR (myErr = eofErr) THEN
      BEGIN                                        {move data into TERec}
         HLock(Handle(myData^^.editRec));
         TESetText(myBuffer, myLength, myData^^.editRec);
         myErr := noErr;
         HUnlock(Handle(myData^^.editRec));
      END;
   DoReadFile := myErr;
END;
```

The `DoReadFile` function takes one parameter specifying the window to read data into. This function first retrieves the handle to that window's document record and extracts the file's reference number from that record. Then `DoReadFile` calls the `SetFPos` function to set the file mark to the beginning of the file having that reference number. There is no need to check that `myFile` has a nonzero value, because `SetFPos` returns an error if you pass it an invalid file reference number.

The second parameter to `SetFPos` specifies the file mark positioning mode; it can contain one of the following values:

```
CONST
   fsAtMark    = 0;  {at current mark}
   fsFromStart = 1;  {set mark relative to beginning of file}
   fsFromLEOF  = 2;  {set mark relative to logical end-of-file}
   fsFromMark  = 3;  {set mark relative to current mark}
```

If you specify `fsAtMark`, the mark is left wherever it's currently positioned, and the third parameter of `SetFPos` is ignored. The next three constants let you position the mark relative to either the beginning of the file, the logical end-of-file, or the current mark. If you specify one of these three constants, the third parameter contains the byte offset (either positive or negative) from the specified point. Here, the appropriate positioning mode is relative to the beginning of the file.

If `DoReadFile` successfully positions the file mark, it next determines the number of bytes in the file by calling the `GetEOF` function. The key step in the `DoReadFile` function is the call to `FSRead`, which reads the specified number of bytes from the file into the specified buffer. In this case, the data is read into a temporary buffer; then the data is moved into the TextEdit record associated with the file. The `FSRead` function returns, in the `myLength` parameter, the number of bytes actually read from the file.

Writing File Data

Generally your application writes data to a file in response to the File menu commands Save or Save As. However, your application might also incorporate a scheme that automatically saves all open documents to disk every few minutes. It therefore makes sense to isolate the routines that handle the menu commands from the routines that

handle the actual writing of data to disk. This section shows how to write the data stored in a TextEdit record to a file. See “Saving a File” on page 1-26 for instructions on handling the Save and Save As menu commands.

It is very easy to write data from a specified buffer into a specified file. You simply position the file mark at the beginning of the file (using `SetFPos`), write the data into the file (using `FSWrite`), and then resize the file to the number of bytes actually written (using `SetEOF`). Listing 1-9 illustrates this sequence.

Listing 1-9 Writing data into a file

```
FUNCTION DoWriteData (myWindow: WindowPtr; myTemp: Integer): OSErr;
VAR
   myData:     MyDocRecHnd;            {handle to a document record}
   myLength:   LongInt;                {number of bytes to write to file}
   myText:     TEHandle;               {handle to TextEdit record}
   myBuffer:   Handle;                 {handle to actual text in TERec}
   myVol:      Integer;                {volume reference number of myFile}
   myErr:      OSErr;
BEGIN
   myData := MyDocRecHnd(GetWRefCon(myWindow)); {get window's data record}
   myText := myData^^.editRec;                  {get TERec}
   myBuffer := myText^^.hText;                  {get text buffer}
   myLength := myText^^.teLength;               {get text buffer size}

   myErr := SetFPos(myTemp, fsFromStart, 0);    {set file mark at start}
   IF myErr = noErr THEN                        {write buffer into file}
      myErr := FSWrite(myTemp, myLength, myBuffer^);
   IF myErr = noErr THEN                        {adjust file size}
      myErr := SetEOF(myTemp, myLength);
   IF myErr = noErr THEN                        {find volume file is on}
      myErr := GetVRefNum(myTemp, myVol);
   IF myErr = noErr THEN                        {flush volume}
      myErr := FlushVol(NIL, myVol);
   IF myErr = noErr THEN                        {show file is up to date}
      myData^^.windowDirty := FALSE;
   DoWriteData := myErr;
END;
```

The `DoWriteData` function first retrieves the TextEdit record attached to the specified window and extracts the address and length of the actual text buffer from that record. Then it calls `SetFPos`, `FSWrite`, and `SetEOF` as just explained. Finally, `DoWriteData` determines the volume containing the file (using the `GetVRefNum` function) and flushes that volume (using the `FlushVol` function). This is necessary to ensure that both the file’s data and the file’s catalog entry are updated.

Notice that the `DoWriteData` function takes a second parameter, `myTemp`, which should be the file reference number of a temporary file, not the file reference number of the file associated with the window whose data you want to write. If you pass the reference number of the file associated with the window, you risk corrupting the file, because the existing file data is overwritten when you position the file mark at the beginning of the file and call `FSWrite`. If `FSWrite` does not complete successfully, it is very likely that the file on disk does not contain the correct document data.

To avoid corrupting the file containing the saved version of a document, always call `DoWriteData` specifying the file reference number of some new, temporary file. Then, when `DoWriteData` completes successfully, you can call the `FSpExchangeFiles` function to swap the contents of the temporary file and the existing file. Listing 1-10 illustrates how to update a file on disk safely; it shows a sequence of updating, renaming, saving, and deleting files that preserves the contents of the existing file until the new version is safely recorded.

Listing 1-10 Updating a file safely

```
FUNCTION DoWriteFile (myWindow): OSErr;
VAR
   myData:     MyDocRecHnd;   {handle to window's document record}
   myFSpec:    FSSpec;        {FSSpec for file to update}
   myTSpec:    FSSpec;        {FSSpec for temporary file}
   myTime:     LongInt;       {current time; for temporary filename}
   myName:     Str255;        {name of temporary file}
   myTemp:     Integer;       {file reference number of temporary file}
   myVRef:     Integer;       {volume reference number of temporary file}
   myDirID:    LongInt;       {directory ID of temporary file}
   myErr:      OSErr;
BEGIN
   myData := MyDocRecHnd(GetWRefCon(myWindow));{get that window's data}
   myFSpec := myData^^.fileFSSpec;           {get FSSpec for existing file}

   GetDateTime(myTime);                      {create a temporary filename}
   NumToString(myTime, myName);

   {Find the temporary folder on file's volume; create it if necessary.}
   myErr := FindFolder(myFSpec.vRefNum, kTemporaryFolderType,
                       kCreateFolder, myVRef, myDirID);
   IF myErr = noErr THEN                     {make an FSSpec for temp file}
      myErr := FSMakeFSSpec(myVRef, myDirID, myName, myTSpec);
   IF (myErr = noErr) OR (myErr = fnfErr) THEN{create a temporary file}
      myErr := FSpCreate(myTSpec, 'trsh', 'trsh', smSystemScript);
   IF myErr = noErr THEN                     {open the newly created file}
```

```
      myErr := FSpOpenDF(myTSpec, fsRdWrPerm, myTemp);
   IF myErr = noErr THEN                    {write data to the data fork}
      myErr := DoWriteData(myWindow, myTemp);
   IF myErr = noErr THEN                    {close the temporary file}
      myErr := FSClose(myTemp);
   IF myErr = noErr THEN                    {swap data in the two files}
      myErr := FSpExchangeFiles(myTSpec, myFSpec);
   IF myErr = noErr THEN                    {delete the temporary file}
      myErr := FSpDelete(myTSpec);
   DoWriteFile := myErr;
END;
```

The essential idea behind this "safe-save" process is to save the data in memory into a new file and then to exchange the contents of the new file and the old version of the file by calling `FSpExchangeFiles`. The `FSpExchangeFiles` function does not move the data on the volume; it merely changes the information in the volume's catalog file and, if the files are open, in their file control blocks (FCBs). The catalog entry for a file contains

- fields that describe the physical data, such as the first allocation block, physical end, and logical end of both the resource and data forks
- fields that describe the file within the file system, such as file ID and parent directory ID

Fields that describe the data remain with the data; fields that describe the file remain with the file. The creation date remains with the file; the modification date remains with the data. (For a more complete description of the `FSpExchangeFiles` function, see the chapter "File Manager" in this book.)

Saving a File

There are several ways for a user to indicate that the current contents of a document should be saved (that is, written to disk). The user can choose the File menu commands Save or Save As, or the user can click the Save button in a dialog box that you display when the user attempts to close a "dirty" document (that is, a document whose contents have changed since the last time it was saved). You can handle the Save menu command quite easily, as illustrated in Listing 1-11.

Listing 1-11 Handling the Save menu command

```
FUNCTION DoSaveCmd: OSErr;
VAR
   myWindow:   WindowPtr;              {pointer to the front window}
   myData:     MyDocRecHnd;            {handle to a document record}
   myErr:      OSErr;
```

```
BEGIN
   myWindow := FrontWindow;            {get front window and its data}
   myData := MyDocRecHnd(GetWRefCon(myWindow));
   IF myData^^.fileRefNum <> 0 THEN    {if window has a file already}
      myErr := DoWriteFile(myWindow);  {then write contents to disk}
   ELSE
      myErr := DoSaveAsCmd;            {else ask for a filename}
   DoSaveCmd := myErr;
END;
```

The DoSaveCmd function simply checks whether the frontmost window is already associated with a file. If so, then DoSaveCmd calls DoWriteFile to write the data to disk (using the "safe-save" process illustrated in the previous section). Otherwise, if no file exists for that window, DoSaveCmd calls DoSaveAsCmd. Listing 1-12 shows a way to define the DoSaveAsCmd function.

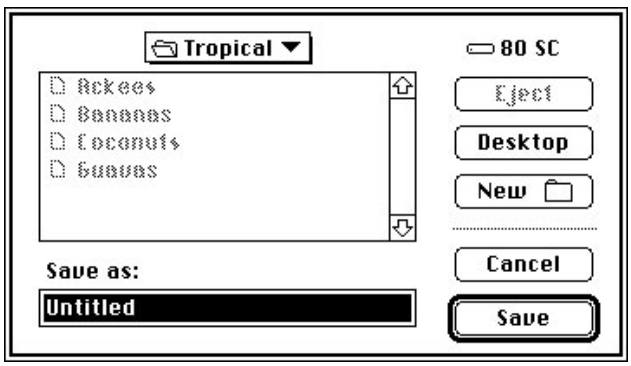
Listing 1-12 Handling the Save As menu command

```
FUNCTION DoSaveAsCmd: OSErr;
VAR
   myWindow:   WindowPtr;          {pointer to the front window}
   myData:     MyDocRecHnd;        {handle to a document record}
   myReply:    StandardFileReply;
   myFile:     Integer;            {file reference number}
   myErr:      OSErr;
BEGIN
   myWindow := FrontWindow;        {get front window and its data}
   myData := MyDocRecHnd(GetWRefCon(myWindow));
   myErr := noErr;

   StandardPutFile('Save as:', 'Untitled', myReply);
   IF myReply.sfGood THEN          {user saves file}
      BEGIN
         IF NOT myReply.sfReplacing THEN
            myErr := FSpCreate(myReply.sfFile, 'MYAP', 'TEXT',
                               smSystemScript);
         IF myErr <> noErr THEN
            Exit(DoSaveAsCmd);
         myData^^.fileFSSpec := myReply.sfFile;

         IF myData^^.fileRefNum <> 0 THEN    {if window already has a file}
            myErr := FSClose(myData^^.fileRefNum);{close it}
```

```
        {Create document's resource fork and copy Finder resources to it.}
        FSpCreateResFile(myData^^.fileFSSpec, 'MYAP', 'TEXT',
                         smSystemScript);
        myErr := ResError;
        IF myErr = noErr THEN
           myFile := FSpOpenResFile(myData^^.fileFSSpec, fsRdWrPerm);
        IF myFile > 0 THEN                  {copy Finder resources}
           myErr := DoCopyResource('STR ', -16396, gAppsResFile, myFile)
        ELSE
           myErr := ResError;
        IF myErr = noErr THEN
           myErr := FSClose(myFile);        {close the resource fork}

        {Open data fork and leave it open.}
        IF myErr = noErr THEN
           myErr := FSpOpenDF(myData^^.fileFSSpec, fsRdWrPerm, myFile);
        IF myErr = noErr THEN
           BEGIN
              myData^^.fileRefNum := myFile;
              SetWTitle(myWindow, myReply.sfFile.name);
              myErr := DoWriteFile(myWindow);
           END;
        DoSaveAsCmd := myErr;
     END;
END;
```

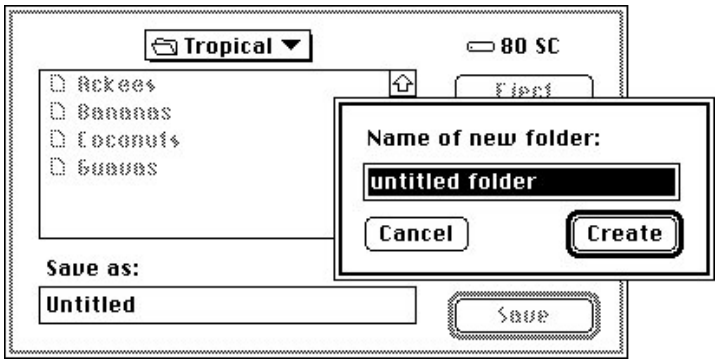
The `StandardPutFile` procedure is similar to the `StandardGetFile` procedure discussed earlier in this chapter. It manages the user interface for the default Save dialog box, illustrated in Figure 1-8.

Figure 1-8 The default Save dialog box



If the user clicks the New Folder button, the Standard File Package presents a subsidiary dialog box like the one shown in Figure 1-9.

Figure 1-9 The new folder dialog box



If the user asks to save a file with a name that already exists at the specified location, the Standard File Package displays a subsidiary dialog box, like the one shown in Figure 1-10, to verify that the new file should replace the existing file.

Figure 1-10 The name conflict dialog box

Note in Listing 1-12 that if the user is not replacing an existing file, the `DoSaveAsCmd` function creates a new file and records the new `FSSpec` record in the window’s document record. Otherwise, if the user is replacing an existing file, `DoSaveAsCmd` simply records, in the window’s document record, the `FSSpec` record returned by `StandardGetFile`.

When `DoSaveAsCmd` creates a new file, it also copies a resource from your application’s resource fork to the resource fork of the newly created file. This resource (with ID –16396) identifies the name of your application. (For more details about this resource, see the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials*.) The `DoSaveAsCmd` function calls the application-defined routine `DoCopyResource`. Listing 1-13 shows a simple way to define the `DoCopyResource` function.

Listing 1-13 Copying a resource from one resource fork to another

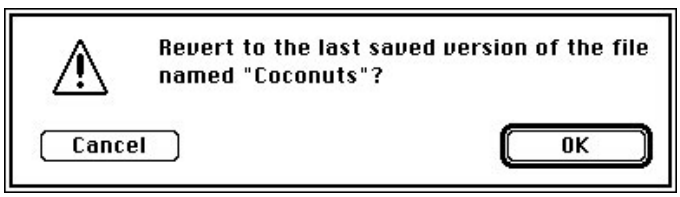
```
FUNCTION DoCopyResource (theType: ResType; theID: Integer;
                         source: Integer; dest: Integer): OSErr;
VAR
   myHandle:   Handle;                        {handle to resource to copy}
   myName:     Str255;                        {name of resource to copy}
   myType:     ResType;                       {ignored; used for GetResInfo}
   myID:       Integer;                       {ignored; used for GetResInfo}
BEGIN
   UseResFile(source);                        {set the source resource file}
   myHandle := GetResource(theType, theID);  {open the source resource}
   IF myHandle <> NIL THEN
      BEGIN
         GetResInfo(myHandle, myID, myType, myName);  {get resource name}
         DetachResource(myHandle);           {detach resource}
         UseResFile(dest);                   {set destination resource file}
         AddResource(myHandle, theType, theID, myName);
         IF ResError = noErr THEN
            WriteResource(myHandle);         {write resource data}
         END;
      DoCopyResource := ResError;               {return result code}
   ReleaseResource(myHandle);
END;
```

See the chapter “Resource Manager” in *Inside Macintosh: More Macintosh Toolbox* for details about the routines used in Listing 1-13.

Reverting to a Saved File

Many applications that manipulate files provide a menu command that allows the user to revert to the last saved version of a document. The technique for handling this command is relatively simple. First you should display a dialog box asking whether to revert to the last saved version of the file, as illustrated in Figure 1-11.

Figure 1-11 A Revert to Saved dialog box



If the user clicks the Cancel button, nothing should happen to the current document. If, however, the user confirms the menu command by clicking OK, you just need to call `DoReadFile` to read the disk version of the file back into the window. Listing 1-14 illustrates how to implement a Revert to Saved menu command.

Listing 1-14 Handling the Revert to Saved menu command

```
FUNCTION DoRevertCmd: OSErr;
VAR
   myWindow:   WindowPtr;          {window for file data}
   myData:     MyDocRecHnd;        {handle to window data}
   myFile:     Integer;            {file reference number}
   myName:     Str255;             {file's name}
   myDialog:   DialogPtr;          {pointer to modal dialog box}
   myItem:     Integer;            {item selected in modal dialog}
   myPort:     GrafPtr;            {the original graphics port}
CONST
   kRevertDialog = 128;            {resource ID of Revert to Saved dialog}
BEGIN
   myWindow := FrontWindow;        {get pointer to front window}
                                   {get handle to window's data record}
   myData := MyDocRecHnd(GetWRefCon(myWindow));
   GetWTitle(myWindow, myName);    {get file's name}
   ParamText(myName, '', '', '');
   myDialog := GetNewDialog(kRevertDialog, NIL, WindowPtr(-1));
   GetPort(myPort);
   SetPort(myDialog);

   REPEAT
      ModalDialog(NIL, myItem);
   UNTIL (myItem = iOK) OR (myItem = iCancel);

   DisposeDialog(myDialog);
   SetPort(myPort);                {restore previous grafPort}

   IF myItem = iOK THEN
      DoRevertCmd := DoReadFile(myWindow);
   ELSE
      DoRevertCmd := noErr;
END;
```

The `DoRevertCmd` function retrieves the document record handle from the frontmost window's reference constant field and then gets the window's title (which is also the name of the file) and inserts it into a modal dialog box.

If the user clicks the OK button, `DoRevertCmd` calls the `DoReadFile` function to read the data from the file into the window. Otherwise, `DoRevertCmd` simply exits without changing the data in the window.

Closing a File

In most cases, your application closes a file after a user clicks in a window's close box or chooses the Close command in the File menu. The Close menu command should be active only when there is actually an active window on the desktop. If there is an active window, you need to determine whether it belongs to your application; if so, you need to handle dialog windows and document windows differently, as illustrated in Listing 1-15.

Listing 1-15 Handling the Close menu command

```
FUNCTION DoCloseCmd: OSErr;
VAR
   myWindow:   WindowPtr;
   myData:     MyDocRecHnd;
   myErr:      OSErr;
BEGIN
   myErr := FALSE;
   myWindow := FrontWindow;         {get window to be closed}
   CASE MyGetWindowType(myWindow) OF
      kDAWindow:
         CloseDeskAcc(WindowPeek(myWindow)^.windowKind);
      kMyModelessDialog:
         HideWindow(myWindow);      {for dialogs, hide the window}
      kMyDocWindow:
         BEGIN
            myData := MyDocRecHnd(GetWRefCon(myWindow));
            myErr := DoCloseFile(myData);
            IF myErr = noErr THEN
               DisposeWindow(myWindow);
         END;
      OTHERWISE
         ;
   END;
   DoCloseCmd := myErr;
END;
```

The `DoCloseCmd` function determines the type of the frontmost window by calling the application-defined function `MyGetWindowType`. (See the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a definition of `MyGetWindowType`.) If the window to be closed is a window belonging to a desk accessory, `DoCloseCmd` closes

the desk accessory. If the window to be closed is a dialog window, this procedure just hides the window. If the window to be closed is a document window, `DoCloseCmd` retrieves its document record handle and calls both `DoCloseFile` (defined in Listing 1-16) and `DisposeWindow`. Before you close the file associated with a window, you should check whether the contents of the window have changed since the last time the document was saved. If so, you should ask the user whether to save those changes. Listing 1-16 illustrates one way to do this.

Listing 1-16 Closing a file

```
FUNCTION DoCloseFile (myData: MyDocRecHnd): OSErr;
VAR
   myErr:      OSErr;
   myDialog:   DialogPtr;        {pointer to modal dialog box}
   myItem:     Integer;          {item selected in alert box}
   myPort:     GrafPtr;          {the original graphics port}
CONST
   kSaveChangesDialog = 129;     {resource of Save changes dialog}
BEGIN
   IF myData^^.windowDirty THEN  {see whether window is dirty}
      BEGIN
         myItem := CautionAlert(kSaveChangesDialog, NIL);
         IF myItem = iCancel THEN{user clicked Cancel}
            BEGIN
               DoCloseFile := usrCanceledErr;
               Exit(DoCloseFile);
            END;
         IF myItem = iSave THEN
            myErr := DoSaveCmd;
      END;
   IF myData^^.fileRefNum <> 0 THEN
      BEGIN
         myErr := FSClose(myData^^.fileRefNum);
         IF myErr = noErr THEN
            BEGIN
               myErr := FlushVol(NIL, myData^^.fileFSSpec.vRefNum);
               myData^^.fileRefNum := 0;  {clear the file reference number}
            END;
      END;
   {Dispose of TextEdit record and controls here (code omitted).}
   DisposeHandle(Handle(myData));         {dispose of document record}
   DoCloseFile := myErr;
END;
```

If the document is an existing file that has not been changed since it was last saved, your application can simply call the `FSClose` function. This routine writes to disk any unwritten data remaining in the volume buffer. The `FSClose` function also updates the information maintained on the volume for that file and removes the access path. The information about the file is not actually written to the disk, however, until the volume is flushed, ejected, or unmounted. To keep the file information current, it's a good idea to follow each call to `FSClose` with a call to the `FlushVol` function.

If the contents of an existing file have been changed, or if a new file is being closed for the first time, your application can call the Dialog Manager routine `CautionAlert` (specifying a resource ID of an `'ALRT'` template) to ask the user whether or not to save the changes. If the user decides not to save the file, you can just call `FSClose` and dispose of the window. Otherwise, `DoCloseFile` calls the `DoSaveCmd` function to save the file to disk.

Opening Files at Application Startup Time

A user often launches your application by double-clicking one of its document icons or by selecting one or more document icons and choosing the Open command in the Finder's File menu. In these cases, your application needs to determine which files the user selected so that it can open each one and display its contents in a window. There are two ways in which your application can determine this.

If the user opens a file from the Finder and if your application supports high-level events, the Finder sends it an Open Documents event. Your application then needs to determine which file or files to open and react accordingly. For a complete description of how to process the Open Documents event, see the chapter "Apple Event Manager" in *Inside Macintosh: Interapplication Communication*.

IMPORTANT

If at all possible, your application should support high-level events. You should use the techniques illustrated in this section only if your application doesn't support high-level events. ▲

If your application does not support high-level events, you need to ask the Finder at application launch time whether or not the user launched the application by selecting some documents. You can do this by calling the `CountAppFiles` procedure and seeing whether the count of files is 1 or more. Then you can call the procedures `GetAppFiles` and `ClrAppFiles` to retrieve the information about the selected files. The technique is illustrated in Listing 1-17.

The `CountAppFiles` procedure determines how many files, if any, the user selected at application startup time. If the value of the `myNum` parameter is nonzero, then `myJob` contains a value that indicates whether the files were selected for opening or printing. Currently, `myJob` can have one of two values:

```
CONST
   appOpen  =  0;    {open the document(s)}
   appPrint =  1;    {print the document(s)}
```

Listing 1-17 Opening files at application launch time

```
PROCEDURE DoInitFiles;
VAR
   myNum:   Integer;    {number of files to be opened or printed}
   myJob:   Integer;    {open or print the files?}
   index:   Integer;    {index of current file}
   myFile:  AppFile;    {file info}
   mySpec:  FSSpec;     {file system specification}
   myErr:   OSErr;
BEGIN
   CountAppFiles(myJob, myNum);
   IF myNum > 0 THEN                        {user selected some files}
      IF myJob = appOpen THEN               {files are to be opened}
         FOR index := 1 TO myNum DO
            BEGIN
               GetAppFiles(index, myFile);   {get file info from Finder}
               myErr := FSMakeFSSpec(myFile.vRefNum, 0, myFile.fName,
                                     mySpec); {make an FSSpec to hold info}
               myErr := DoOpenFile(mySpec);  {read in file's data}
               ClrAppFiles(index);           {show we've got the info}
            END;
END;
```

In Listing 1-17, if the files are to be opened, then `DoInitFiles` obtains information about them by calling the `GetAppFiles` procedure for each one. The `GetAppFiles` procedure returns the information in a record of type `AppFile`.

```
TYPE AppFile   =
   RECORD
      vRefNum:    Integer;    {working directory reference number}
      fType:      OSType;     {file type}
      versNum:    Integer;    {version number; ignored}
      fName:      Str255;     {filename}
   END;
```

Because the function `DoOpenFile` takes an `FSSpec` record as a parameter, `DoInitFiles` next converts the information returned in the `myFile` parameter into an `FSSpec` record, using `FSMakeFSSpec`. Then `DoInitFiles` calls `DoOpenFile` to read the file data and `ClrAppFiles` to let the Finder know that it has processed the information for that file.

Note
The `vRefNum` field of an `AppFile` record does not contain a volume reference number; instead it contains a working directory reference number, which encodes both the volume reference number and the parent directory ID. (That's why the second parameter passed to `FSMakeFSSpec` in Listing 1-17 is 0.) ◆

Using a Preferences File

Many applications allow the user to alter various settings that control the operation or configuration of the application. For example, your application might allow the user to specify the size and placement of any new windows or the default font used to display text in those windows. You can create a preferences file in which to record user preferences, and your application can retrieve that file whenever it is launched.

In deciding how to structure your preferences file, it is important to distinguish document-specific settings from application-specific settings. Some user-specifiable settings affect only a particular document. For example, the user might have changed the text font in a particular window. When you save the text in the window, you also want to save the current font setting. Generally you can do this by storing the font name in a resource in the document file's resource fork. Then, when the user opens that document again, you check for the presence of such a resource, retrieve the information stored in it, and set the document font accordingly.

Some settings, such as a default text font, are not specific to a particular document. You might store such settings in the application's resource fork, but generally it is better to store them in a separate preferences file. The main reason for this is to avoid problems that can arise if an application is located on a server volume. If preferences are stored in resources in the application's resource fork, those preferences apply to all users executing that application. Worse yet, the resources can become corrupted if several different users attempt to alter the settings at the same time.

Thus, it is best to store application-specific settings in a preferences file. The Operating System provides a special folder in the System Folder, called Preferences, where you can store that file. Listing 1-18 illustrates a way to open your application's preferences file.

Listing 1-18 Opening a preferences file

```
PROCEDURE DoGetPreferences;
VAR
   myErr:      OSErr;
   myVRef:     Integer; {volume ref num of Preferences folder}
   myDirID:    LongInt; {dir ID of Preferences folder}
   mySpec:     FSSpec;  {FSSpec for the preferences file}
   myName:     Str255;  {name of the application}
   myRef:      Integer; {ref num of app's resource file; ignored}
   myHand:     Handle;  {handle to Finder information; ignored}
   myRefNum:   Integer; {file reference number}
```

```
CONST
   kPrefID = 128;       {resource ID of STR# with filename}
BEGIN
   {Determine the name of the preferences file.}
   GetIndString(myName, kPrefID, 1);

   {Find the Preferences folder in the System Folder.}
   myErr := FindFolder(kOnSystemDisk, kPreferencesFolderType,
                       kDontCreateFolder, myVRef, myDirID);
   IF myErr = noErr THEN
      myErr := FSMakeFSSpec(myVRef, myDirID, myName, mySpec);
   IF myErr = noErr THEN
      myRefNum := FSpOpenResFile(mySpec, fsCurPerm);

   {Read your preference settings here.}

   CloseResFile(myRefNum);
END;
```

The `DoGetPreferences` procedure first determines the name of the preferences file it is to open and read. To allow easy localization, you should store the name in a resource of type `'STR#'` in your application's resource file. The `DoGetPreferences` procedure assumes that the name is stored as the first string in the resource having ID `kPrefID`.

The technique shown here assumes that your preference settings can all be stored in resources. As a result, Listing 1-18 calls the Resource Manager function `FSpOpenResFile` to open the resource fork of your preferences file. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for complete details on opening resource files and reading resources from them.

Adjusting the File Menu

Your application should dim any File menu commands that are not available at the time the user pulls down the File menu. For example, if your application does not yet have a document window open, then the Save, Save As, and Revert commands should be dimmed. You can adjust the File menu easily using the technique shown in Listing 1-19.

Listing 1-19 Adjusting the File menu

```
PROCEDURE DoAdjustFileMenu;
VAR
   myWindow:   WindowPtr;
   myMenu:     MenuHandle;
   myData:     MyDocRecHnd;              {handle to window data}
```

```
BEGIN
   myWindow := FrontWindow;
   IF myWindow = NIL THEN
      BEGIN
         myMenu := GetMHandle(mFile);
         DisableItem(myMenu, iSave);      {disable Save}
         DisableItem(myMenu, iSaveAs);    {disable Save As}
         DisableItem(myMenu, iRevert);    {disable Revert}
         DisableItem(myMenu, iClose);     {disable Close}
      END
   ELSE IF MyGetWindowType(myWindow) = kMyDocWindow THEN
      BEGIN
         myData := MyDocRecHnd(GetWRefCon(myWindow));
         myMenu := GetMHandle(mFile);
         EnableItem(myMenu, iSaveAs);     {enable Save As}
         EnableItem(myMenu, iClose);      {enable Close}

         IF myData^^.windowDirty THEN
            BEGIN
               EnableItem(myMenu, iSave);       {enable Save}
               EnableItem(myMenu, iRevert);     {enable Revert}
            END
         ELSE
            BEGIN
               DisableItem(myMenu, iSave);      {disable Save}
               DisableItem(myMenu, iRevert);    {disable Revert}
            END;
      END;
END;
```

Your application should call `DoAdjustFileMenu` whenever it receives a mouse-down event in the menu bar. (No doubt you want to include code appropriate for enabling and disabling other menu items too.) See the chapter "Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for details on the menu enabling and disabling procedures used in Listing 1-19.

File Management Reference

This section describes the data structures and routines used in this chapter to illustrate basic file management operations. The section "Data Structures" shows the Pascal data structures for the file system specification record and the standard file reply record. The sections that follow describe the Standard File Package routines for opening and saving

documents and the File Manager routines for accessing files, manipulating files and directories, accessing volumes, and getting information about documents to be opened when your application is launched.

For a description of other file-related data structures and routines, see the chapters "File Manager" and "Standard File Package" in this book.

Data Structures

This section describes the data structures that your application can use to exchange information with the File Manager and the Standard File Package. The techniques described in this chapter use file system specification records and standard file reply records.

File System Specification Record

The file system specification record for files and directories is defined by the `FSSpec` data type.

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

Field descriptions

`vRefNum` The volume reference number of the volume containing the specified file or directory.

`parID` The directory ID of the directory containing the specified file or directory.

`name` The name of the specified file or directory.

Standard File Reply Records

The procedures `StandardGetFile` and `StandardPutFile` both return information to your application using a standard file reply record, which is defined by the `StandardFileReply` data type. The reply record identifies selected files with a file system specification record, which you can pass directly to many of the File Manager functions described in the sections that follow. The reply record also contains fields that support several Finder features.

```
TYPE StandardFileReply =
RECORD
   sfGood:        Boolean;    {TRUE if user did not cancel}
   sfReplacing:   Boolean;    {TRUE if replacing file with same name}
   sfType:        OSType;     {file type}
   sfFile:        FSSpec;     {selected file, folder, or volume}
   sfScript:      ScriptCode; {script of file, folder, or volume name}
   sfFlags:       Integer;    {Finder flags of selected item}
   sfIsFolder:    Boolean;    {selected item is a folder}
   sfIsVolume:    Boolean;    {selected item is a volume}
   sfReserved1:   LongInt;    {reserved}
   sfReserved2:   Integer;    {reserved}
END;
```

Field descriptions

`sfGood` Reports whether the reply record is valid. The value is `TRUE` after the user clicks Save or Open; `FALSE` after the user clicks Cancel. When the user has completed the dialog box, the other fields in the reply record are valid only if the `sfGood` field contains `TRUE`.

`sfReplacing` Reports whether a file to be saved replaces an existing file of the same name. This field is valid only after a call to the `StandardPutFile` or `CustomPutFile` procedure. When the user assigns a name that duplicates that of an existing file, the Standard File Package asks for verification by displaying a subsidiary dialog box (illustrated in Figure 1-10). If the user verifies the name, the Standard File Package sets the `sfReplacing` field to `TRUE` and returns to your application; if the user cancels the overwriting of the file, the Standard File Package returns to the main dialog box. If the name does not conflict with an existing name, the Standard File Package sets the field to `FALSE` and returns.

`sfType` Contains the file type of the selected file. (File types are described in the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.) Only `StandardGetFile` and `CustomGetFile` return a file type in this field.

`sfFile` Describes the selected file, folder, or volume with a file system specification record, which contains a volume reference number, parent directory ID, and name. (See the chapter "File Manager" in this book for a complete description of the file system specification record.) If the selected item is an alias for another item, the Standard File Package resolves the alias and places the file system specification record for the target in the `sfFile` field when the user completes the dialog box. If the selected file is a stationery pad, the reply record describes the file itself, not a copy of the file.

`sfScript` Identifies the script in which the name of the document is to be displayed. (This information is used by the Finder and by the Standard File Package.) A script code of `smSystemScript` (–1) represents the default system script.

`sfFlags`	Contains the Finder flags from the Finder information record in the catalog entry for the selected file. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a description of the Finder flags.) This field is returned only by `StandardGetFile` and `CustomGetFile`. If your application supports stationery, it should check the stationery bit in the Finder flags to determine whether to treat the selected file as stationery. Unlike the Finder, the Standard File Package does not automatically create a document from a stationery pad and pass your application the new document. If the user opens a stationery document from within an application that does not support stationery, the Standard File Package displays a dialog box warning the user that the master copy is being opened.
`sfIsFolder`	Reports whether the selected item is a folder (`TRUE`) or a file or volume (`FALSE`). This field is meaningful only during the execution of a dialog hook function.
`sfIsVolume`	Reports whether the selected item is a volume (`TRUE`) or a file or folder (`FALSE`). This field is meaningful only during the execution of a dialog hook function.
`sfReserved1`	Reserved.
`sfReserved2`	Reserved.

Application Files Records

The `GetAppFiles` procedure returns information about files opened at application launch time in an application files record, defined by the `AppFile` data type:

```
TYPE AppFile   =
RECORD
   vRefNum:    Integer;    {working directory reference number}
   fType:      OSType;     {file type}
   versNum:    Integer;    {version number; ignored}
   fName:      Str255;     {filename}
END;
```

Field descriptions

`vRefNum`	A working directory reference number that encodes the volume and parent directory of the file.
`fType`	The file type.
`versNum`	Reserved.
`fName`	The filename.

File Specification Routines

If your application has no special user interface requirements, you can use the `StandardGetFile` and `StandardPutFile` procedures to display the default dialog boxes for opening and saving documents. For a description of more advanced file specification routines, see the chapter "Standard File Package" in this book.

StandardGetFile

You can use the `StandardGetFile` procedure to display the default Open dialog box when the user is opening a file.

```
PROCEDURE StandardGetFile (fileFilter: FileFilterProcPtr;
                           numTypes: Integer;
                           typeList: SFTypeList;
                           VAR reply: StandardFileReply);
```

`fileFilter` A pointer to an optional file filter function, provided by your application, through which `StandardGetFile` passes files of the specified types.

`numTypes` The number of file types to be displayed. If you specify a `numTypes` value of –1, the first filtering passes files of all types.

`typeList` A list of file types to be displayed.

`reply` The reply record, which `StandardGetFile` fills in before returning.

DESCRIPTION

The `StandardGetFile` procedure presents a dialog box through which the user specifies the name and location of a file to be opened. While the dialog box is active, `StandardGetFile` gets and handles events until the user completes the interaction, either by selecting a file to open or by canceling the operation. `StandardGetFile` returns the user's input in a record of type `StandardFileReply`.

The `fileFilter`, `numTypes`, and `typeList` parameters together determine which files appear in the displayed list. The first filtering is by file type, which you specify in the `numTypes` and `typeList` parameters. The `numTypes` parameter specifies the number of file types to be displayed. You can specify one or more types. If you specify a `numTypes` value of –1, the first filtering passes files of all types.

The `fileFilter` parameter points to an optional file filter function, provided by your application, through which `StandardGetFile` passes files of the specified types. See the chapter "Standard File Package" in this book for a complete description of how you specify this filter function.

SPECIAL CONSIDERATIONS

The `StandardGetFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `StandardGetFile` is available before calling it.

Because `StandardGetFile` may move memory, you should not call it at interrupt time.

StandardPutFile

You can use the `StandardPutFile` procedure to display the default Save dialog box when the user is saving a file.

```
PROCEDURE StandardPutFile (prompt: Str255; defaultName: Str255;
                           VAR reply: StandardFileReply);
```

`prompt` The prompt message to be displayed over the text field.

`defaultName` The initial name of the file.

`reply` The reply record, which `StandardPutFile` fills in before returning.

DESCRIPTION

The `StandardPutFile` procedure presents a dialog box through which the user specifies the name and location of a file to be written to. The dialog box is centered on the screen. While the dialog box is active, `StandardPutFile` gets and handles events until the user completes the interaction, either by selecting a name and authorizing the save or by canceling the save. The `StandardPutFile` procedure returns the user's input in a record of type `StandardFileReply`.

SPECIAL CONSIDERATIONS

The `StandardPutFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `StandardPutFile` is available before calling it.

Because `StandardPutFile` may move memory, you should not call it at interrupt time.

File Access Routines

This section describes the File Manager's file access routines. When you call one of these routines, you specify a file by a path reference number (which the File Manager returns to your application when your application opens the file). Unless your application has very specialized needs, you should be able to manage all file access (for example, writing data to the file) using the routines described in this section. Typically you use these routines to operate on a file's data fork, but in certain circumstances you might want to use them on a file's resource fork as well.

Reading, Writing, and Closing Files

You can use the functions `FSRead`, `FSWrite`, and `FSClose` to read data from a file, write data to a file, and close an open file. All three of these functions operate on open files. You can use any one of a variety of routines to open a file (for example, `FSpOpenDF`).

FSRead

You can use the `FSRead` function to read any number of bytes from an open file.

```
FUNCTION FSRead (refNum: Integer; VAR count: LongInt;
                 buffPtr: Ptr): OSErr;
```

`refNum`	The file reference number of an open file.
`count`	On input, the number of bytes to read; on output, the number of bytes actually read.
`buffPtr`	A pointer to the data buffer into which the bytes are to be read.

DESCRIPTION

The `FSRead` function attempts to read the requested number of bytes from the specified file into the specified buffer. The `buffPtr` parameter points to that buffer; this buffer is allocated by your application and must be at least as large as the `count` parameter.

Because the read operation begins at the current mark, you might want to set the mark first by calling the `SetFPos` function. If you try to read past the logical end-of-file, `FSRead` reads in all the data up to the end-of-file, moves the mark to the end-of-file, and returns `eofErr` as its function result. Otherwise, `FSRead` moves the file mark to the byte following the last byte read and returns `noErr`.

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`eofErr`	–39	Logical end-of-file reached
`posErr`	–40	Attempt to position mark before start of file
`fLckdErr`	-45	File is locked
`paramErr`	–50	Negative `count`
`rfNumErr`	–51	Bad reference number
`afpAccessDenied`	–5000	User does not have the correct access to the file

FSWrite

You can use the `FSWrite` function to write any number of bytes to an open file.

```
FUNCTION FSWrite (refNum: Integer; VAR count: LongInt;
                  buffPtr: Ptr): OSErr;
```

`refNum` The file reference number of an open file.

`count` On input, the number of bytes to write to the file; on output, the number of bytes actually written.

`buffPtr` A pointer to the data buffer from which the bytes are to be written.

DESCRIPTION

The `FSWrite` function takes the specified number of bytes from the specified data buffer and attempts to write them to the specified file. Because the write operation begins at the current mark, you might want to set the mark first by calling the `SetFPos` function.

If the write operation completes successfully, `FSWrite` moves the file mark to the byte following the last byte written and returns `noErr`. If you try to write past the logical end-of-file, `FSWrite` moves the logical end-of-file. If you try to write past the physical end-of-file, `FSWrite` adds one or more clumps to the file and moves the physical end-of-file accordingly.

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Disk full
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`posErr`	–40	Attempt to position mark before start of file
`wPrErr`	–44	Hardware volume lock
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Software volume lock
`paramErr`	–50	Negative `count`
`rfNumErr`	–51	Bad reference number
`wrPermErr`	–61	Read/write permission doesn't allow writing

FSClose

You can use the `FSClose` function to close an open file.

```
FUNCTION FSClose (refNum: Integer): OSErr;
```

`refNum` The file reference number of an open file.

DESCRIPTION

The `FSClose` function removes the access path for the specified file and writes the contents of the volume buffer to the volume.

Note

The `FSClose` function calls `PBFlushFile` internally to write the file's bytes onto the volume. To ensure that the file's catalog entry is updated, you should call `FlushVol` after you call `FSClose`. ◆

▲ **WARNING**

Make sure that you do not call `FSClose` with a file reference number of a file that has already been closed. Attempting to close the same file twice may result in loss of data on a volume. See the description of file control blocks in the chapter "File Manager" in this book for a discussion of how this can happen. ▲

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`fnfErr`	–43	File not found
`rfNumErr`	–51	Bad reference number

Manipulating the File Mark

You can use the functions `GetFPos` and `SetFPos` to get or set the current position of the file mark.

GetFPos

You can use the `GetFPos` function to determine the current position of the mark before reading from or writing to an open file.

```
FUNCTION GetFPos (refNum: Integer; VAR filePos: LongInt): OSErr;
```

`refNum`	The file reference number of an open file.
`filePos`	On output, the current position of the mark.

DESCRIPTION

The `GetFPos` function returns, in the `filePos` parameter, the current position of the file mark for the specified open file. The position value is zero-based; that is, the value of `filePos` is 0 if the file mark is positioned at the beginning of the file.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
fnOpnErr	–38	File not open
rfNumErr	–51	Bad reference number
gfpErr	–52	Error during GetFPos

SetFPos

You can use the SetFPos function to set the position of the file mark before reading from or writing to an open file.

```
FUNCTION SetFPos (refNum: Integer; posMode: Integer;
                  posOff: LongInt): OSErr;
```

refNum	The file reference number of an open file.
posMode	The positioning mode.
posOff	The positioning offset.

DESCRIPTION

The SetFPos function sets the file mark of the specified file. The posMode parameter indicates how to position the mark; it must contain one of the following values:

```
CONST
   fsAtMark    = 0;  {at current mark}
   fsFromStart = 1;  {set mark relative to beginning of file}
   fsFromLEOF  = 2;  {set mark relative to logical end-of-file}
   fsFromMark  = 3;  {set mark relative to current mark}
```

If you specify fsAtMark, the mark is left wherever it's currently positioned, and the posOff parameter is ignored. The next three constants let you position the mark relative to either the beginning of the file, the logical end-of-file, or the current mark. If you specify one of these three constants, you must also pass in posOff a byte offset (either positive or negative) from the specified point. If you specify fsFromLEOF, the value in posOff must be less than or equal to 0.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
fnOpnErr	–38	File not open
eofErr	–39	Logical end-of-file reached
posErr	–40	Attempt to position mark before start of file
rfNumErr	–51	Bad reference number

Manipulating the End-of-File

You can use the functions `GetEOF` and `SetEOF` to get or set the logical end-of-file of an open file.

GetEOF

You can use the `GetEOF` function to determine the current logical end-of-file of an open file.

```
FUNCTION GetEOF (refNum: Integer; VAR logEOF: LongInt): OSErr;
```

`refNum` The file reference number of an open file.

`logEOF` On output, the logical end-of-file.

DESCRIPTION

The `GetEOF` function returns, in the `logEOF` parameter, the logical end-of-file of the specified file.

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`rfNumErr`	–51	Bad reference number
`afpAccessDenied`	–5000	User does not have the correct access to the file

SEE ALSO

For a description of the logical and physical end-of-file, see the section "File Access Characteristics" on page 1-8.

SetEOF

You can use the `SetEOF` function to set the logical end-of-file of an open file.

```
FUNCTION SetEOF (refNum: Integer; logEOF: LongInt): OSErr;
```

`refNum` The file reference number of an open file.

`logEOF` The logical end-of-file.

DESCRIPTION

The `SetEOF` function sets the logical end-of-file of the specified file. If you attempt to set the logical end-of-file beyond the physical end-of-file, the physical end-of-file is set 1 byte beyond the end of the next free allocation block; if there isn't enough space on the volume, no change is made, and `SetEOF` returns `dskFulErr` as its function result.

If you set the `logEOF` parameter to 0, all space occupied by the file on the volume is released. The file still exists, but it contains 0 bytes. Setting a file fork's end-of-file to 0 is therefore not the same as deleting the file (which removes both file forks at once).

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Disk full
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`wPrErr`	–44	Hardware volume lock
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Software volume lock
`rfNumErr`	–51	Bad reference number
`wrPermErr`	–61	Read/write permission doesn't allow writing

SEE ALSO

For a description of the logical and physical end-of-file, see the section "File Access Characteristics" on page 1-8.

File and Directory Manipulation Routines

The File Manager includes a set of file and directory manipulation routines that accept `FSSpec` records as parameters. Depending on the requirements of your application and on the environment in which it is running, you may be able to accomplish all your file and directory operations by using these routines.

Before calling any of these routines, however, you should call the `Gestalt` function to ensure that they are available in the operating environment. (See "Testing for File Management Routines" on page 1-14 for an illustration of calling `Gestalt`.) If these routines are not available, you can call the corresponding HFS routines.

Opening, Creating, and Deleting Files

The File Manager provides the `FSpOpenDF`, `FSpCreate`, and `FSpDelete` routines, which allow you to open, create, and delete files.

FSpOpenDF

You can use the FSpOpenDF function to open a file's data fork.

```
FUNCTION FSpOpenDF (spec: FSSpec; permission: SignedByte;
                    VAR refNum: Integer): OSErr;
```

`spec` An FSSpec record specifying the file whose data fork is to be opened.

`permission`
A constant indicating the desired file access permissions.

`refNum` A reference number of an access path to the file's data fork.

DESCRIPTION

The FSpOpenDF function opens the data fork of the file specified by the `spec` parameter and returns a file reference number in the `refNum` parameter. You can pass that reference number as a parameter to any of the low- or high-level file access routines.

The `permission` parameter specifies the kind of access permission mode you want. You can specify one of these constants:

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

In most cases, you can simply set the permission parameter to `fsCurPerm`. Some applications request `fsRdWrPerm`, to ensure that they can both read from and write to a file.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`tmfoErr`	–42	Too many files open
`fnfErr`	–43	File not found
`opWrErr`	–49	File already open for writing
`permErr`	–54	Attempt to open locked file for writing
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access to the file

FSpCreate

You can use the `FSpCreate` function to create a new file.

```
FUNCTION FSpCreate (spec: FSSpec; creator: OSType;
                    fileType: OSType; scriptTag: ScriptCode):
                    OSErr;
```

`spec` An `FSSpec` record specifying the file to be created.

`creator` The creator of the new file.

`fileType` The file type of the new file.

`scriptTag` The code of the script system in which the filename is to be displayed. If you have established the name and location of the new file using either the `StandardPutFile` or `CustomPutFile` procedure, specify the script code returned in the reply record. (See the chapter "Standard File Package" in this book for a description of `StandardPutFile` and `CustomPutFile`.) Otherwise, specify the system script by setting the `scriptTag` parameter to the value `smSystemScript`.

DESCRIPTION

The `FSpCreate` function creates a new file (both forks) with the specified type, creator, and script code. The new file is unlocked and empty. The date and time of creation and last modification are set to the current date and time.

See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on file types and creators.

Files created using `FSpCreate` are not automatically opened. If you want to write data to the new file, you must first open the file using a file access routine (such as `FSpOpenDF`).

Note

The resource fork of the new file exists but is empty. You'll need to call one of the Resource Manager procedures `CreateResFile`, `HCreateResFile`, or `FSpCreateResFile` to create a resource map in the file before you can open it (by calling one of the Resource Manager functions `OpenResFile`, `HOpenResFile`, or `FSpOpenResFile`). ◆

RESULT CODES

noErr	0	No error
dirFulErr	–33	File directory full
dskFulErr	–34	Disk is full
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	Directory not found or incomplete pathname
wPrErr	–44	Hardware volume lock
vLckdErr	–46	Software volume lock
dupFNErr	–48	Duplicate filename and version
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access
afpObjectTypeErr	–5025	A directory exists with that name

FSpDelete

You can use the `FSpDelete` function to delete files and directories.

```
FUNCTION FSpDelete (spec: FSSpec): OSErr;
```

`spec` An `FSSpec` record specifying the file or directory to delete.

DESCRIPTION

The `FSpDelete` function removes a file or directory. If the specified target is a file, both forks of the file are deleted. The file ID reference, if any, is removed.

A file must be closed before you can delete it. Similarly, a directory must be empty before you can delete it. If you attempt to delete an open file or a nonempty directory, `FSpDelete` returns the result code `fBsyErr`. `FSpDelete` also returns the result code `fBsyErr` if the directory has an open working directory associated with it.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
fBsyErr	–47	File busy, directory not empty, or working directory control block open
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access

Exchanging the Data in Two Files

The function `FSpExchangeFiles` allows you to exchange the data in two files.

FSpExchangeFiles

You can use the `FSpExchangeFiles` function to exchange the data stored in two files on the same volume.

```
FUNCTION FSpExchangeFiles (source: FSSpec; dest: FSSpec): OSErr;
```

`source` The source file. The contents of this file and its file information are placed in the file specified by the `dest` parameter.

`dest` The destination file. The contents of this file and its file information are placed in the file specified by the `source` parameter.

DESCRIPTION

The `FSpExchangeFiles` function swaps the data in two files by changing the information in the volume's catalog and, if the files are open, in the file control blocks. You should use `FSpExchangeFiles` when updating an existing file, so that the file ID remains valid in case the file is being tracked through its file ID. The `FSpExchangeFiles` function changes the fields in the catalog entries that record the location of the data and the modification dates. It swaps both the data forks and the resource forks.

The `FSpExchangeFiles` function works on both open and closed files. If either file is open, `FSpExchangeFiles` updates any file control blocks associated with the file. Exchanging the contents of two files requires essentially the same access permissions as opening both files for writing.

The files whose data is to be exchanged must both reside on the same volume. If they do not, `FSpExchangeFiles` returns the result code `diffVolErr`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`ioErr`	–36	I/O error
`fnfErr`	–43	File not found
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Volume is locked or read-only
`paramErr`	–50	Function not supported by volume
`volOfflinErr`	–53	Volume is offline
`wrgVolTypErr`	–123	Not an HFS volume
`diffVolErr`	–1303	Files on different volumes
`afpAccessDenied`	–5000	User does not have the correct access
`afpObjectTypeErr`	–5025	Object is a directory, not a file
`afpSameObjectErr`	–5038	Source and destination files are the same

Creating File System Specifications

The FSMakeFSSpec function allows you to create FSSpec records.

FSMakeFSSpec

You can use the FSMakeFSSpec function to initialize an FSSpec record to particular values for a file or directory.

```
FUNCTION FSMakeFSSpec (vRefNum: Integer; dirID: LongInt;
                       fileName: Str255; VAR spec: FSSpec):
                       OSErr;
```

vRefNum — A volume specification. This parameter can contain a volume reference number, a working directory reference number, a drive number, or 0 (to specify the default volume).

dirID — A directory specification. This parameter usually specifies the parent directory ID of the target object. If the directory is sufficiently specified by either the vRefNum or fileName parameter, dirID can be set to 0. If you explicitly specify dirID (that is, if it has any value other than 0), and if vRefNum specifies a working directory reference number, dirID overrides the directory ID included in vRefNum. If the fileName parameter contains an empty string, FSMakeFSSpec creates an FSSpec record for a directory specified by either the dirID or vRefNum parameter.

fileName — A full or partial pathname. If fileName specifies a full pathname, FSMakeFSSpec ignores both the vRefNum and dirID parameters. A partial pathname might identify only the final target, or it might include one or more parent directory names. If fileName specifies a partial pathname, then vRefNum, dirID, or both must be valid.

spec — A file system specification to be filled in by FSMakeFSSpec.

DESCRIPTION

The FSMakeFSSpec function fills in the fields of the spec parameter using the information contained in the other three parameters. Call FSMakeFSSpec whenever you want to create an FSSpec record.

You can pass the input to FSMakeFSSpec in several ways. The chapter “File Manager” in this book explains how FSMakeFSSpec interprets its input.

If the specified volume is mounted and the specified parent directory exists, but the target file or directory doesn’t exist in that location, FSMakeFSSpec fills in the record and then returns fnfErr instead of noErr. The record is valid, but it describes a target that doesn’t exist. You can use the record for other operations, such as creating a file with the FSpCreate function.

In addition to the result codes that follow, `FSMakeFSSpec` can return a number of other File Manager error codes. If your application receives any result code other than `noErr` or `fnfErr`, all fields of the resulting `FSSpec` record are set to 0.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume doesn't exist
`fnfErr`	–43	File or directory does not exist (`FSSpec` is still valid)

Volume Access Routines

This section describes the high-level volume access routines. Unless your application has very specialized needs, you should be able to manage all volume access using the routines described in this section. In fact, most applications are likely to need only the `FlushVol` function described in the next section, "Updating Volumes."

When you call one of these routines, you specify a volume by a volume reference number (which you can obtain, for example, by calling the `GetVInfo` function, or from the reply record returned by the Standard File Package). You can also specify a volume by name, but this is generally discouraged, because there is no guarantee that volume names are unique.

Updating Volumes

When you close a file, you should call `FlushVol` to ensure that any changed contents of the file are written to the volume.

FlushVol

You can use the `FlushVol` function to write the contents of the volume buffer and update information about the volume.

```
FUNCTION FlushVol (volName: StringPtr; vRefNum: Integer): OSErr;
```

`volName` A pointer to the name of a mounted volume.

`vRefNum` A volume reference number, a working directory reference number, a drive number, or 0 for the default volume.

DESCRIPTION

On the specified volume, the `FlushVol` function writes the contents of the associated volume buffer and descriptive information about the volume (if they've changed since the last time `FlushVol` was called). This information is written to the volume.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad volume name
paramErr	–50	No default volume
nsDrvErr	–56	No such drive

Obtaining Volume Information

You can get information about a volume by calling the `GetVInfo` or `GetVRefNum` function.

GetVInfo

You can use the `GetVInfo` function to get information about a mounted volume.

```
FUNCTION GetVInfo (drvNum: Integer; volName: StringPtr;
                   VAR vRefNum: Integer;
                   VAR freeBytes: LongInt): OSErr;
```

`drvNum`	The drive number of the volume for which information is requested.
`volName`	On output, a pointer to the name of the specified volume.
`vRefNum`	The volume reference number of the specified volume.
`freeBytes`	The available space (in bytes) on the specified volume.

DESCRIPTION

The `GetVInfo` function returns the name, volume reference number, and available space (in bytes) for the specified volume. You specify a volume by providing its drive number in the `drvNum` parameter. You can pass 0 in the `drvNum` parameter to get information about the default volume.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
paramErr	–50	No default volume

GetVRefNum

You can use the `GetVRefNum` function to get a volume reference number from a file reference number.

```
FUNCTION GetVRefNum (refNum: Integer; VAR vRefNum: Integer):
                     OSErr;
```

`refNum` The file reference number of an open file.

`vRefNum` On exit, the volume reference number of the volume containing the file specified by `refNum`.

DESCRIPTION

The `GetVRefNum` function returns the volume reference number of the volume containing the specified file. If you also want to determine the directory ID of the specified file's parent directory, call the `PBGetFCBInfo` function.

RESULT CODES

`noErr`	0	No error
`rfNumErr`	–51	Bad reference number

Application Launch File Routines

You can call `GetAppParms` to determine your application's name and the reference number of its resource file. When your application starts up, you can call `CountAppFiles` to determine whether the user selected any documents to open or print. If so, you can call `GetAppFiles` and `ClrAppFiles` to process the information passed to your application by the Finder.

Note

If your application supports high-level events, you receive this information from the Finder in an Open Documents or Print Documents event. ◆

GetAppParms

You can use the `GetAppParms` procedure to get information about the current application and about files selected by the user for opening or printing.

```
PROCEDURE GetAppParms(VAR apName: Str255; VAR apRefNum: Integer;
                      VAR apParam: Handle);
```

`apName` On output, the name of the calling application.

`apRefNum` On output, the reference number of the application's resource file.

`apParam` On output, a handle to the Finder information about files to open or print.

DESCRIPTION

The `GetAppParms` procedure returns information about the current application. You can call `GetAppParms` at application launch time to determine which files, if any, the user has selected in the Finder for opening or printing. You can call `GetAppParms` at any time to determine the current application's name and the reference number of the application's resource fork.

The `GetAppParms` procedure returns the application's name in the `apName` parameter and the reference number of its resource fork in the `apRefNum` parameter. A handle to the Finder information is returned in `apParam`. This information consists of a word that encodes the message or action to be performed, a word that indicates how many files to process, and a list of Finder information about each such file. The Finder information has the structure of an `AppFile` record, except that the filename occupies only as many bytes as are required to hold the name (padded to an even number of bytes, if necessary). In general, it is easier to use the `GetAppFiles` procedure to access the Finder information.

SPECIAL CONSIDERATIONS

If you simply want to determine the application's resource file reference number, you can call the Resource Manager function `CurResFile` when your application starts up.

If you need more extensive information about the application than `GetAppParms` provides, you can use the Process Manager function `GetCurrentProcess`.

ASSEMBLY-LANGUAGE INFORMATION

You can get the application's name, reference number, and handle to the Finder information directly from the global variables `CurApName`, `CurApRefNum`, and `AppParmHandle`.

CountAppFiles

You can use the `CountAppFiles` procedure to determine how many documents (if any) the user has selected at application launch time for opening or printing.

```
PROCEDURE CountAppFiles (VAR message: Integer;
                         VAR count: Integer);
```

`message` The action to be performed on the selected files.

`count` The number of files selected.

DESCRIPTION

The `CountAppFiles` procedure deciphers the Finder information passed to your application and returns information about the files that were selected when your application was started up. On exit, the `count` parameter contains the number of selected files, and the `message` parameter contains an integer that indicates whether the files are to be opened or printed. The `message` parameter contains one of these constants:

```
CONST
   appOpen  =  0;    {open the document(s)}
   appPrint =  1;    {print the document(s)}
```

GetAppFiles

You can use the `GetAppFiles` procedure to retrieve information about each file selected at application startup for opening or printing.

```
PROCEDURE GetAppFiles (index: Integer; VAR theFile: AppFile);
```

`index` The index of the file whose information is returned.

`theFile` A structure containing the returned information.

DESCRIPTION

The `GetAppFiles` procedure returns information about a file that was selected when your application was started up (as listed in the Finder information). The `index` parameter indicates the file for which information should be returned; it must be between 1 and the number returned by `CountAppFiles`, inclusive.

ClrAppFiles

You can use the `ClrAppFiles` procedure to notify the Finder that you have processed the information about a file selected for opening or printing at application startup.

```
PROCEDURE ClrAppFiles (index: Integer);
```

`index` The index of the file whose information is to be cleared.

DESCRIPTION

The `ClrAppFiles` procedure changes the Finder information passed to your application about the specified file so that the Finder knows you've processed the file. The `index` parameter must be between 1 and the number returned by `CountAppFiles`, inclusive. You should call `ClrAppFiles` for every document your application opens or prints, so that the information returned by `CountAppFiles` and `GetAppFiles` is always correct. The `ClrAppFiles` procedure sets the file type in the Finder information to 0.

Summary of File Management

Pascal Summary

Constants

```
CONST
   {Gestalt constants}
   gestaltFSAttr           =  'fs  ';  {file system attributes selector}
   gestaltHasFSSpecCalls   =  1;       {supports FSSpec records}
   gestaltStandardFileAttr =  'stdf';  {Standard File attributes selector}
   gestaltStandardFile58   =  0;       {supports StandardPutFile etc.}
   gestaltFindFolderAttr   =  'fold';  {FindFolder attributes selector}
   gestaltFindFolderPresent=  0;       {FindFolder is present}

   {access modes for opening files}
   fsCurPerm         =  0;       {whatever permission is allowed}
   fsRdPerm          =  1;       {read permission}
   fsWrPerm          =  2;       {write permission}
   fsRdWrPerm        =  3;       {exclusive read/write permission}
   fsRdWrShPerm      =  4;       {shared read/write permission}

   {file mark positioning modes}
   fsAtMark          =  0;       {at current mark}
   fsFromStart       =  1;       {set mark relative to beginning of file}
   fsFromLEOF        =  2;       {set mark relative to logical end-of-file}
   fsFromMark        =  3;       {set mark relative to current mark}
   rdVerify          =  64;      {add to above for read-verify}

   {messages from CountAppFiles}
   appOpen           =  0;       {open the document(s)}
   appPrint          =  1;       {print the document(s)}
```

Data Types

File System Specification Record

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

   FSSpecPtr         =  ^FSSpec;
   FSSpecHandle      =  ^FSSpecPtr;
```

Standard File Reply Record

```
TYPE StandardFileReply=
   RECORD
      sfGood:        Boolean;    {TRUE if user did not cancel}
      sfReplacing:   Boolean;    {TRUE if replacing file with same name}
      sfType:        OSType;     {file type}
      sfFile:        FSSpec;     {selected item}
      sfScript:      ScriptCode; {script of selected item's name}
      sfFlags:       Integer;    {Finder flags of selected item}
      sfIsFolder:    Boolean;    {selected item is a folder}
      sfIsVolume:    Boolean;    {selected item is a volume}
      sfReserved1:   LongInt;    {reserved}
      sfReserved2:   Integer;    {reserved}
   END;
```

Application Files Record

```
TYPE AppFile         =
   RECORD
      vRefNum:       Integer;    {working directory reference number}
      fType:         OSType;     {file type}
      versNum:       Integer;    {version number; ignored}
      fName:         Str255;     {filename}
   END;

   SFTypeList        =  ARRAY[0..3] OF OSType;

   FileFilterProcPtr =  ProcPtr; {file filter function}
```

File Specification Routines

Opening Files

```
PROCEDURE StandardGetFile   (fileFilter: FileFilterProcPtr;
                             numTypes: Integer; typeList: SFTypeList;
                             VAR reply: StandardFileReply);
```

Saving Files

```
PROCEDURE StandardPutFile   (prompt: Str255; defaultName: Str255;
                             VAR reply: StandardFileReply);
```

File Access Routines

Reading, Writing, and Closing Files

```
FUNCTION FSRead             (refNum: Integer; VAR count: LongInt;
                             buffPtr: Ptr): OSErr;
FUNCTION FSWrite            (refNum: Integer; VAR count: LongInt;
                             buffPtr: Ptr): OSErr;
FUNCTION FSClose            (refNum: Integer): OSErr;
```

Manipulating the File Mark

```
FUNCTION GetFPos            (refNum: Integer; VAR filePos: LongInt): OSErr;
FUNCTION SetFPos            (refNum: Integer; posMode: Integer;
                             posOff: LongInt): OSErr;
```

Manipulating the End-of-File

```
FUNCTION GetEOF             (refNum: Integer; VAR logEOF: LongInt): OSErr;
FUNCTION SetEOF             (refNum: Integer; logEOF: LongInt): OSErr;
```

File and Directory Manipulation Routines

Opening, Creating, and Deleting Files

```
FUNCTION FSpOpenDF          (spec: FSSpec; permission: SignedByte;
                             VAR refNum: Integer): OSErr;
FUNCTION FSpCreate          (spec: FSSpec; creator: OSType;
                             fileType: OSType; scriptTag: ScriptCode):
                             OSErr;
FUNCTION FSpDelete          (spec: FSSpec): OSErr;
```

Exchanging the Data in Two Files

```
FUNCTION FSpExchangeFiles   (source: FSSpec; dest: FSSpec): OSErr;
```

Creating File System Specifications

```
FUNCTION FSMakeFSSpec       (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255; VAR spec: FSSpec): OSErr;
```

Volume Access Routines

Updating Volumes

```
FUNCTION FlushVol           (volName: StringPtr; vRefNum: Integer): OSErr;
```

Obtaining Volume Information

```
FUNCTION GetVInfo           (drvNum: Integer; volName: StringPtr;
                             VAR vRefNum: Integer; VAR freeBytes: LongInt):
                             OSErr;
FUNCTION GetVRefNum         (refNum: Integer; VAR vRefNum: Integer): OSErr;
```

Application Launch File Routines

```
PROCEDURE GetAppParms       (VAR apName: Str255; VAR apRefNum: Integer;
                             VAR apParam: Handle);
PROCEDURE CountAppFiles     (VAR message: Integer; VAR count: Integer);
PROCEDURE GetAppFiles       (index: Integer; VAR theFile: AppFile);
PROCEDURE ClrAppFiles       (index: Integer);
```

C Summary

Constants

```
/*Gestalt constants*/
#define gestaltFSAttr            'fs  '   /*file system attributes selector*/
#define gestaltFullExtFSDispatching 0     /*exports HFSDispatch traps*/
#define gestaltHasFSSpecCalls       1     /*supports FSSpec records*/
#define gestaltFindFolderAttr    'fold'   /*FindFolder attributes selector*/
#define gestaltFindFolderPresent    0     /*FindFolder is present*/
```

```
/*Gestalt Standard File attributes selector and reply*/
#define gestaltStandardFileAttr  'stdf'
#define gestaltStandardFile58       0

/*values for requesting file read/write permissions*/
enum {
   fsCurPerm            = 0,     /*whatever permission is allowed*/
   fsRdPerm             = 1,     /*read permission*/
   fsWrPerm             = 2,     /*write permission*/
   fsRdWrPerm           = 3,     /*exclusive read/write permission*/
   fsRdWrShPerm         = 4};    /*shared read/write permission*/

/*file mark positioning modes*/
enum {
   fsAtMark             = 0,     /*at current mark}
   fsFromStart          = 1,     /*set mark relative to beginning of file*/
   fsFromLEOF           = 2,     /*set mark relative to logical end-of-file*/
   fsFromMark           = 3,     /*set mark relative to current mark*/
   rdVerify             = 64};   /*add to above for read-verify*/

/*messages from CountAppFiles*/
enum {
   appOpen              =  0,    /*open the document(s)*/
   appPrint             =  1};   /*print the document(s)*/
```

Data Types

File System Specification Record

```
struct FSSpec  {                    /*file system specification*/
      short       vRefNum;          /*volume reference number*/
      long        parID;            /*directory ID of parent directory*/
      Str63       name;             /*filename or directory name*/
};

typedef struct FSSpec FSSpec;
typedef FSSpec *FSSpecPtr;
typedef FSSpecPtr *FSSpecHandle;
```

Standard File Reply Record

```
struct StandardFileReply {       /*enhanced standard file reply record*/
      Boolean        sfGood;     /*TRUE if user did not cancel*/
      Boolean        sfReplacing;/*TRUE if replacing file with same name*/
      OSType         sfType;     /*file type*/
      FSSpec         sfFile;     /*selected file, folder, or volume*/
      ScriptCode     sfScript;   /*script of file, folder, or volume name*/
      short          sfFlags;    /*Finder flags of selected item*/
      Boolean        sfIsFolder; /*selected item is a folder*/
      Boolean        sfIsVolume; /*selected item is a volume*/
      long           sfReserved1;/*reserved*/
      short          sfReserved2;/*reserved*/
};

typedef struct StandardFileReply StandardFileReply;
```

Application Files Record

```
struct AppFile {
      short          vRefNum;    /*working directory reference number*/
      OSType         fType;      /*file type*/
      short          versNum;    /*version number; ignored*/
      Str255         fName;      /*filename*/
   END;

typedef struct AppFile AppFile;
```

Standard File Type List

```
typedef OSType SFTypeList[4];
```

Callback Routine Pointer Types

```
/*file filter function*/
typedef pascal Boolean (*FileFilterProcPtr)
                           (ParmBlkPtr PB);
```

File Specification Routines

Opening Files

```
pascal void StandardGetFile (const Str255 prompt,
                             FileFilterProcPtr fileFilter,
                             short numTypes, SFTypeList typeList,
                             StandardFileReply *reply);
```

Saving Files

```
pascal void StandardPutFile (const Str255 prompt, const Str255 defaultName,
                             StandardFileReply *reply);
```

File Access Routines

Reading, Writing, and Closing Files

```
pascal OSErr FSRead         (short refNum, long *count, Ptr buffPtr);
pascal OSErr FSWrite        (short refNum, long *count, Ptr buffPtr);
pascal OSErr FSClose        (short refNum);
```

Manipulating the File Mark

```
pascal OSErr GetFPos        (short refNum, long *filePos);
pascal OSErr SetFPos        (short refNum, short posMode, long posOff);
```

Manipulating the End-of-File

```
pascal OSErr GetEOF         (short refNum, long *logEOF);
pascal OSErr SetEOF         (short refNum, long logEOF);
```

File and Directory Manipulation Routines

Opening, Creating, and Deleting Files

```
pascal OSErr FSpOpenDF      (const FSSpec *spec, char permission,
                             short *refNum);
pascal OSErr FSpCreate      (const FSSpec *spec, OSType creator,
                             OSType fileType, ScriptCode scriptTag);
pascal OSErr FSpDelete      (const FSSpec *spec);
```

Exchanging the Data in Two Files

```
pascal OSErr FSpExchangeFiles
                            (const FSSpec *source, const FSSpec *dest);
```

Creating File System Specifications

```
pascal OSErr FSMakeFSSpec   (short vRefNum, long dirID,
                             ConstStr255Param fileName, FSSpecPtr spec);
```

Volume Access Routines

Updating Volumes

```
pascal OSErr FlushVol        (StringPtr volName, short vRefNum);
```

Obtaining Volume Information

```
pascal OSErr GetVInfo       (short drvNum, StringPtr volName,
                             short *vRefNum, long *freeBytes);
pascal OSErr GetVRefNum     (short refNum, short *vRefNum);
```

Application Launch File Routines

```
pascal void GetAppParms     (Str255 apName, short *apRefNum,
                             Handle *apParam);
pascal void CountAppFiles   (short *message, short *count);
pascal void GetAppFiles     (short index, AppFile *theFile);
pascal void ClrAppFiles     (short index);
```

Assembly-Language Summary

Global Variables

`AppParmHandle`	long	Handle to Finder information.
`CurApName`	32 bytes	Name of current application (length byte followed by up to 31 characters).
`CurApRefNum`	word	Reference number of current application's resource file.

Result Codes

noErr	0	No error
dirFulErr	–33	File directory full
dskFulErr	–34	All allocation blocks on the volume are full
nsvErr	–35	Volume not found
ioErr	–36	I/O error
bdNamErr	–37	Bad filename or volume name
fnOpnErr	–38	File not open
eofErr	–39	Logical end-of-file reached
posErr	–40	Attempt to position mark before start of file
tmfoErr	–42	Too many files open
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File locked
vLckdErr	–46	Software volume lock
fBsyErr	–47	File is busy; one or more files are open; directory not empty or working directory control block is open
dupFNErr	–48	A file with the specified name and version number already exists
opWrErr	–49	File already open for writing
paramErr	–50	Parameter error
rfNumErr	–51	Reference number specifies nonexistent access path
gfpErr	–52	Error during GetFPos
volOfflinErr	–53	Volume is offline
permErr	–54	Attempt to open locked file for writing
nsDrvErr	–56	Specified drive number doesn't match any number in the drive queue
wrPermErr	–61	Read/write permission doesn't allow writing
dirNFErr	–120	Directory not found or incomplete pathname
wrgVolTypErr	–123	Not an HFS volume
notAFileErr	–1302	Specified file is a directory
diffVolErr	–1303	Files are on different volumes
sameFileErr	–1306	Source and destination files are the same
afpAccessDenied	–5000	User does not have the correct access to the file
afpObjectTypeErr	–5025	Object is a directory, not a file; a directory exists with that name
afpSameObjectErr	–5038	Source and destination files are the same

File Manager

Contents

This chapter describes how your application can use the File Manager to store and access data in files or to manipulate files, directories, and volumes. It also provides a complete description of all File Manager routines, data types, and constants.

You need to read the information in this chapter if you wish to use File Manager routines other than those described in the chapter "Introduction to File Management" earlier in this book. That chapter shows how to use the File Manager, the Standard File Package, and other system software components to handle the typical File menu commands and perform other common file-manipulation operations. This chapter addresses a number of other important file-related issues, including

- using the low-level File Manager routines
- locking and unlocking byte ranges in shared files
- searching a volume for files or directories satisfying certain criteria
- obtaining information about files, directories, and volumes

This chapter also addresses some advanced topics of interest primarily to designers of very specialized applications or file-system utility programs. These advanced topics include

- how the File Manager organizes file and directory data on disk
- how the File Manager organizes information in memory

To use this chapter, you should already be familiar with the information presented in the chapter "Introduction to File Management" earlier in this book.

This chapter begins with a general introduction to the File Manager and the services it provides. Then it describes

- ways of identifying files, directories, and volumes
- file access permissions
- directory access privileges
- running in a shared environment

About the File Manager

The File Manager is the part of the Macintosh Operating System that manages the organization, reading, and writing of data located on physical data storage devices such as disk drives. This data includes the data in documents as well as other collections of data used to maintain the hierarchical file system (HFS) and other system software services. To accomplish these tasks, the File Manager interacts with many other components of the system software. For example, the Resource Manager uses File Manager routines when it needs to read and write resource data. Similarly, the File Manager calls the Device Manager to perform the actual reading and writing of data on a physical data storage device. In general, you'll use the Resource Manager to read and write data in a file's resource fork and the File Manager to read and write data in a file's data fork. You'll also use the File Manager to perform operations on directories and volumes.

The File Manager provides a large number of routines for performing various operations on files, directories, and volumes. The requirements of your application will dictate which of these routines you will need to use. Many applications simply need to open files, read and write the data in those files, and then close the files. Other applications might provide more capabilities, such as the ability to copy a file or move a file to another directory. A few file-system utilities perform even more extensive file operations and hence need to use some of the advanced routines provided by the File Manager. For example, a disk scavenger might need to make a byte-by-byte search through a volume to find pieces of a deleted file.

You can often use one of several File Manager routines to accomplish a particular task. This is because many of the File Manager routines are provided in two different forms: high level and low level. The low-level routines generally provide the greatest control over the requested task; they are identified by the prefixes PB and PBH, indicating that they take the address of a parameter block as a parameter. The high-level routines are always defined in terms of low-level routines; they are identified by prefixes such as FSp or H, indicating how you identify files or directories using those routines, or by no special prefix at all.

You pass information to a high-level routine using the routine's parameters. A high-level routine has as many parameters as are necessary to pass the information it requires.

You pass information to a low-level routine by filling in fields in a parameter block and then passing the address of the parameter block to the routine. In all cases, a low-level routine uses more fields in the parameter block than there are parameters in the corresponding high-level routine. As a result, you can use those low-level routines to perform more advanced operations or to provide more extensive information than you can with the corresponding high-level routines. This is the principal reason you might choose to use a low-level routine instead of its corresponding high-level routine.

IMPORTANT

If you use the low-level File Manager routines, be sure to clear all unused fields of the parameter block. ▲

Low-level routines also accept a parameter indicating whether you want the routine to be executed synchronously or asynchronously. If you request synchronous execution, control does not return to your application until the routine has been executed. This allows you to inspect the routine's result code to see whether the routine was successfully completed. If so, your application can continue by performing other operations that depend on the successful completion of that routine.

If you request asynchronous execution, an **I/O request** is put into the **file I/O queue** and control returns to your application immediately—possibly even before the actual I/O operation is completed. The File Manager takes requests from the queue one at a time and processes them; meanwhile, your application is free to work on other things. Routines that are executed asynchronously return control to your application with the result code `noErr` as soon as the call is placed in the file I/O queue. Return of control does not signal successful completion of the call, but simply successful queuing of the request. To determine when the call is actually completed, you can poll the `ioResult` field of the parameter block. This field is set to a positive number when the call is made

and set to the actual result code when the call is completed. If necessary, you can also install a **completion routine** that is executed when the asynchronous call is completed. See "Completion Routines" on page 2-240 for details about completion routines.

Note
Although you can request asynchronous execution for most low-level routines, the device driver for the device on which the target file, directory, or volume resides might not support asynchronous operations. For example, the current implementation of the SCSI Manager allows synchronous execution only. The Sony disk driver and AppleShare server software do, however, support asynchronous operation. ◆

The following sections describe the various capabilities of the File Manager. For full details on any of the routines mentioned in these sections, see the descriptions given in "File Manager Reference" beginning on page 2-87.

File Manipulation

The File Manager provides a number of routines that allow you to manipulate files. You can open a file fork, read and write the data in it, adjust its logical end-of-file, set the file mark, allocate blocks to a file, and close a file.

To manipulate the data in a file, you first need to open the file. You can open a file using one of several routines, depending on whether you want to use low-level or high-level routines and how you identify the file to open. Table 2-1 lists the file-opening routines.

Table 2-1 Routines for opening file forks

FSSpec	HFS High-Level	HFS Low-Level	Description
FSpOpenDF	HOpenDF	PBHOpenDF	Open a file's data fork.
FSpOpenRF	HOpenRF	PBHOpenRF	Open a file's resource fork.
	HOpen	PBHOpen	Open a driver or file data fork.

All the high-level FSSpec routines require you to specify a file using a file system specification record. All the HFS routines, whether high or low level, require you to specify a file by its volume, directory, and name.

No matter which routine you use to open a file, you need to specify a **file permission** that governs the kind of access your application can have to that file. You can specify one of these constants:

```
CONST
   fsCurPerm      =  0;    {whatever permission is allowed}
   fsRdPerm       =  1;    {read permission}
```

```
fsWrPerm       =  2;    {write permission}
fsRdWrPerm     =  3;    {exclusive read/write permission}
fsRdWrShPerm   =  4;    {shared read/write permission}
```

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

Use the constant `fsRdPerm` to request permission to read the file. Similarly, use the constant `fsWrPerm` to request permission to write to the file. If write permission is granted, no other access paths are granted write permission. Note, however, that the File Manager does not support write-only access to a file. As a result, `fsWrPerm` is synonymous with `fsRdWrPerm`.

There are two types of read / write permission—exclusive and shared. Often you want exclusive read / write permission, so that users can safely read and alter portions of a file. If your application requests and is granted exclusive read / write permission, no users are granted permission to write to the file; other users may, however, be granted permission to read the file.

Shared read / write permission allows multiple access paths for writing and reading. It is safe to have multiple read / write paths open to a file only if there is some way of locking a portion of the file before writing to that portion of the file. You can use the File Manager functions `PBLockRange` and `PBUnlockRange` to lock and unlock ranges of bytes in a file. These functions, however, are supported only on remotely mounted volumes or on local volumes that are sharable on the network. As a result, you should request shared read / write permission only if range locking is available. See “Shared File Access Permissions” on page 2-15 for details on permissions in shared environments.

Note

Don’t assume that successfully opening a file for writing ensures that you can actually write data to the file. The File Manager allows you to open with write permission a file located on a locked volume, and you won’t receive an error until you first try to write data to the file. To be safe, you can call the `PBHGetVInfo` function to make sure that the volume is writable. ◆

When you successfully open a file fork, you receive a **file reference number** that uniquely identifies the open file. You can pass that number to the File Manager routines that allow you to manipulate open files. Table 2-2 lists the routines that operate on open files.

The File Manager provides a number of routines that allow you to operate on files that are closed. You can create, delete, get and set information, and lock and unlock files. You can also move files within a volume and exchange data in two files. Table 2-2 lists these routines.

Table 2-2 Routines for operating on open file forks

High-Level	Low-Level	Description
FSRead	PBRead	Read bytes from an open file fork.
FSWrite	PBWrite	Write bytes to an open file fork.
FSClose	PBClose	Close an open file fork.
GetFPos	PBGetFPos	Get the position of the file mark.
SetFPos	PBSetFPos	Set the position of the file mark.
GetEOF	PBGetEOF	Get the current logical end-of-file.
SetEOF	PBSetEOF	Set the current logical end-of-file.
Allocate	PBAllocate	Add allocation blocks to a file fork.
AllocContig	PBAllocContig	Add contiguous allocation blocks to a file fork.
	PBFlushFile	Update the disk contents of a file fork.
GetVRefNum		Get volume reference number of an open file.

Table 2-3 Routines for operating on closed files

FSSpec	HFS High-Level	HFS Low-Level	Description
FSpCreate	HCreate	PBHCreate	Create both forks of a new file.
FSpDelete	HDelete	PBHDelete	Delete both forks of a file.
FSpGetFInfo	HGetFInfo	PBHGetFInfo	Get a file's Finder information.
FSpSetFInfo	HSetFInfo	PBHSetFInfo	Set a file's Finder information.
FSpSetFLock	HSetFLock	PBHSetFLock	Lock a file.
FSpRstFLock	HRstFLock	PBHRstFLock	Unlock a file.
FSpCatMove	CatMove	PBCatMove	Move a file or directory within a volume.
FSpRename	HRename	PBHRename	Rename a file or directory.
		PBGetCatInfo	Get information about a file or directory.
		PBSetCatInfo	Set information about a file or directory.

Note
You can use the functions listed in Table 2-2 on open files as well, except for those functions that create or delete file forks. ◆

You can exchange the data in two files using the `FSpExchangeFiles` and `PBExchangeFiles` functions. If you need to create a file system specification record, you can use the `FSMakeFSSpec` or `PBMakeFSSpec` function.

Directory Manipulation

The File Manager provides a number of routines that allow you to manipulate directories. For example, you can create and delete directories, get information about a directory, and move and rename directories. The directory manipulation routines are listed in Table 2-2.

Table 2-4 Routines for operating on directories

FSSpec	HFS High-Level	HFS Low-Level	Description
`FSpDirCreate`	`DirCreate`	`PBDirCreate`	Create a directory.
`FSpDelete`	`HDelete`	`PBHDelete`	Delete a directory.
`FSpGetFInfo`	`HGetFInfo`	`PBHGetFInfo`	Get Finder information for a directory.
`FSpSetFInfo`	`HSetFInfo`	`PBHSetFInfo`	Set Finder information for a directory.
`FSpSetFLock`	`HSetFLock`	`PBHSetFLock`	Lock a directory.
`FSpRstFLock`	`HRstFLock`	`PBHRstFLock`	Unlock a directory.
`FSpCatMove`	`CatMove`	`PBCatMove`	Move a file or directory within a volume.
`FSpRename`	`HRename`	`PBHRename`	Rename a file or directory.
		`PBGetCatInfo`	Get information about a file or directory.
		`PBSetCatInfo`	Set information about a file or directory.

The File Manager includes a number of routines that allow you to manipulate working directories. See Table 2-2. Most applications do not need to use working directories.

Table 2-5 Routines for manipulating working directories

High-Level	Low-Level	Description
`OpenWD`	`PBOpenWD`	Open a working directory.
`CloseWD`	`PBCloseWD`	Close a working directory.
`GetWDInfo`	`PBGetWDInfo`	Get information about a working directory.

Volume Manipulation

The File Manager provides a number of routines that allow you to manipulate volumes. For example, you can obtain information about a mounted volume, update the information on a volume, unmount a mounted volume or place it offline, and so forth. Most applications don't need explicit access to volumes. The Standard File Package and the Finder handle most events related to the insertion and ejection of disks.

When the Event Manager function `WaitNextEvent` (or `GetNextEvent`) receives a disk-inserted event, it calls the Desk Manager function `SystemEvent`. The Desk Manager in turn calls the File Manager function `PBMountVol`, which attempts to mount the volume on the disk. The result of the `PBMountVol` call is put into the high-order word of the event message, and the drive number is put into its low-order word. If the result code indicates that an error occurred, you need to call the Disk Initialization Manager routine `DIBadMount` to allow the user to initialize or eject the volume. For details, see the chapter "Disk Initialization Manager" in this book.

After a volume has been mounted, your application can call `GetVInfo`, which returns the name, the amount of unused space, and the volume reference number. Given a file reference number, you can get the volume reference number of the volume containing that file by calling either `GetVRefNum` or `GetFCBInfo`.

You can unmount or place offline any volumes that aren't currently being used. To unmount a volume, call `UnmountVol`, which flushes a volume (by calling `FlushVol`) and releases all of the memory it uses. To place a volume offline, call `PBOffLine`, which flushes a volume and releases all of the memory used for it except for the volume control block. The File Manager places offline volumes online as needed, but your application must remount any unmounted volumes it wants to access. The File Manager itself may place volumes offline during its normal operation.

Note
If you make a call to an offline volume, the File Manager displays the disk switch dialog box and waits for the user to reinsert the disk containing the volume. When the user inserts the required disk, the File Manager mounts the volume and then reissues your original call. To avoid presenting the user with numerous disk switch dialog boxes, you might need to check that a volume is online before attempting to access data on it. ◆

To protect against data loss due to power interruption or unexpected disk ejection, you should periodically call `FlushVol` (probably after each time you close a file), which writes the contents of the volume buffer and all access path buffers (if any) to the volume and updates the descriptive information contained on the volume.

Whenever your application is finished with a disk, or when the user chooses Eject from a menu, call the `Eject` function. This function calls `FlushVol`, places the volume offline, and then physically ejects the volume from its drive.

If you would like all File Manager calls to apply to a particular volume, specify it as the default volume. You can use the `HGetVol` (or `GetVol`) function to determine the name and volume reference number of the default volume, and the `SetVol` function to make any mounted volume the default.

Normally, volume initialization and naming are handled by the Disk Initialization Manager. If you want to initialize a volume explicitly or erase all files from a volume, you can call the Disk Initialization Manager directly. When you want to change the name of a volume, call the `HRename` function.

Table 2-6 summarizes the volume-manipulation routines. Most of these routines require you to specify a volume either by name or by volume reference number.

Table 2-6 Routines for operating on volumes

High-Level	Low-Level	Description
	`PBMountVol`	Mount a volume.
`UnmountVol`	`PBUnmountVol`	Unmount a volume.
`Eject`	`PBEject`	Eject a volume.
	`PBOffLine`	Place a volume offline.
`FlushVol`	`PBFlushVol`	Update a volume.
`GetVol`	`PBGetVol`	Get the default volume.
`HGetVol`	`PBHGetVol`	Get the default volume.
`SetVol`	`PBSetVol`	Set the default volume.
`HSetVol`	`PBHSetVol`	Set the default volume.
`GetVInfo`	`PBHGetVInfo`	Get information about a volume.
	`PBSetVInfo`	Set information about a volume.
	`PBHGetVolParms`	Determine capabilities of a volume.
	`PBCatSearch`	Search a volume for files or directories satisfying certain criteria.

Volume Searching

The File Manager provides several routines that you can use to search a volume for files or directories having specific characteristics. For example, you can search for all files with modification dates of two days ago or less or all directories with the string “Temp” in their names.

In general, you should avoid searching entire volumes, because a search of large volumes can consume significant amounts of time. Suppose you are looking for a particular file (for example, a dictionary file against which your application needs to check the spelling of a document). In this case, you can save time and increase the chances of finding the correct file by storing and later resolving an alias record that describes the desired file. See the chapter “Alias Manager” in this book for details on using alias records.

Alternatively, suppose you need to find the location of a standard system directory, such as the Preferences folder or the Temporary Items folder. To perform this search most efficiently, you should use the `FindFolder` function. See the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials* for details.

In some cases, however, you do need to search volumes. For instance, a backup utility needs to search an entire volume to find which files and directories, if any, might need to be backed up. In these cases, you can choose either of two general search strategies: you can search the volume’s catalog by calling the `PBCatSearch` function, or you can use a recursive, indexed search by calling the `PBGetCatInfo` function (see Table 2-2).

Table 2-7 Routines for manipulating working directories

Routine	Description
`PBCatSearch`	Search a volume’s catalog file for files or directories.
`PBGetCatInfo`	Get information about a single catalog file entry.

Using the `PBCatSearch` function is the fastest and most reliable way to search the catalog file of an HFS volume for files and directories satisfying certain criteria. The `PBCatSearch` function returns a list of `FSSpec` records describing the files or directories that match the criteria specified by your application.

However, `PBCatSearch` is not available on all volumes or in all versions of the File Manager. See “Determining the Features of the File Manager” on page 2-33 for instructions on how to determine whether the system software and the target volume both support the `PBCatSearch` function.

Note

The `PBCatSearch` function is available on all volumes that support the AppleTalk Filing Protocol (AFP) version 2.1. This includes volumes and directories shared using the file sharing software introduced in system software version 7.0 and using the AppleShare 3.0 file server software. ◆

In environments where `PBCatSearch` is not available, you'll need to do a search that recursively descends the directory hierarchy and reads through the catalog entries of all files and directories located in each directory in that hierarchy. You can do this by making indexed calls to the `PBGetCatInfo` function, which is supported by all system software versions and by all volumes. However, using this recursive, indexed search method is usually significantly slower than using the `PBCatSearch` function. (For example, a recursive, indexed search that takes over 6 minutes might take about 20 seconds using `PBCatSearch`.)

See "Searching a Volume" beginning on page 2-39 for examples of using both `PBCatSearch` and `PBGetCatInfo` to search a volume for files and directories.

Shared Environments

Any operating environment that supports multiple users and multiple access to data or applications is known as a **shared environment.** A shared environment can be a number of workstations attached to a network as well as a single workstation executing a multi-user operating system such as A/UX.

The File Manager supports access both to locally mounted volumes and to volumes located on devices attached to remote machines on a network. For example, AppleShare, Apple's file-server application, allows users to share data, applications, and disk storage over a network. System software version 7.0 introduced File Sharing, a local version of AppleShare that allows users to make some or all of the files on a volume available over the network. To do so, a user establishes a volume or directory as a **share point,** making it available for use by registered users or guests on the network.

It is a virtual certainty that some users will run your application in a shared environment. The File Manager, Chooser, and other system software components cooperate to make access to remote volumes largely transparent to your application. As a result, most applications do not need to accommodate shared environments explicitly. You can read and write files, for instance, regardless of whether they are located on a local or a remote volume.

If your application performs certain operations on files, however, you might be able to save considerable time by using special shared environment routines. Suppose, for example, that you want to copy a file to another directory on a volume. In the general case, you handle this by reading a buffer of data from the source file and then writing it to the destination file. If the source and destination volumes are remote, however, this technique might involve the copying of a lot of data over the network. To optimize remote file copying, the File Manager provides the `PBHCopyFile` function, which copies a remote file without sending the data across the network. Similarly, the `PBHMoveRename` function allows you to move and optionally rename a file located on a remote volume.

The File Manager provides routines that allow you to control other aspects of a shared environment, including

- providing multiple users with shared read/write access to files
- locking and unlocking byte ranges within a file to ensure exclusive access to data during updates

- enabling and disabling sharing on local volumes and directories
- getting and setting access privileges for directories
- determining volume mounting and login information so that any volume can be unmounted and remounted easily

Table 2-8 lists the File Manager routines that you can use in a shared environment. Note that all of these are low-level routines.

Table 2-8 Shared environment routines

Routine	Description
`PBHOpenDeny`	Open a file's data fork using the access deny modes.
`PBHOpenRFDeny`	Open a file's resource fork using the access deny modes.
`PBLockRange`	Lock a portion of a shared file.
`PBUnlockRange`	Unlock a previously locked portion of a shared file.
`PBShare`	Establish a volume or directory as a share point.
`PBUnshare`	Remove a share point from a shared environment.
`PBGetUGEntry`	Get a list of users and groups on the local file server.
`PBHGetDirAccess`	Get the access control information for a directory.
`PBHSetDirAccess`	Set the access control information for a directory.
`PBGetVolMountInfoSize`	Get the size of a volume mounting information record.
`PBGetVolMountInfo`	Get volume mounting information.
`PBVolumeMount`	Mount a volume.
`PBHGetLogInInfo`	Get the method used to log on to a shared volume.
`PBHMapID`	Get the name of a user or group from its ID.
`PBHMapName`	Get the ID of a user or group from its name.
`PBHCopyFile`	Copy a file on a remote volume.
`PBHMoveRename`	Move (and perhaps rename) a file on a remote volume.

The following sections describe the capabilities provided by these routines.

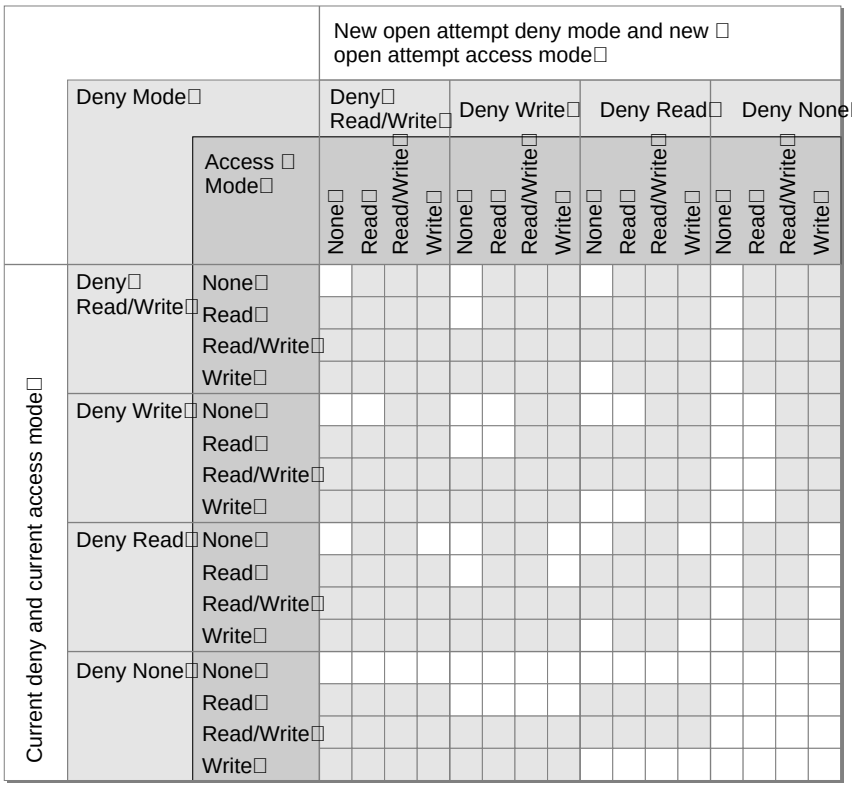
Shared File Access Permissions

In a shared environment, files can be shared at a file or subfile level. At a file level, a project schedule could be read by many users simultaneously but updated by only one user at a time. At a subfile level, different records of a data base file could be updated by several users at the same time.

The access modes provided by the standard file-opening routines prove insufficient for shared files. Two additional open functions, PBHOpenDeny and PBHOpenRFDeny, allow the ability to *deny* access as well. These **deny modes** are cumulative, combining to determine the current access permissions for a file. For instance, if the first opening routine denies reading to others and the second denies writing, both reading and writing are then denied for the file.

Figure 2-1 shows how new access and deny modes are granted or refused according to a file's current access and deny modes. An unshaded square indicates that a new open call with the listed permissions would succeed; otherwise, the new open call would fail.

Figure 2-1 Access and deny mode synchronization



You specify deny modes by setting bits in the ioDenyModes field of the parameter block passed to PBHOpenDeny or PBHOpenRFDeny. Currently four bits of this field are meaningful:

Bit	Meaning
0	If set, request read permission
1	If set, request write permission
4	If set, deny other users read permission to this file
5	If set, deny other users write permission to this file

The combination of access and deny requests allows four common opening possibilities:

- **Browsing access.** You request browsing access by specifying both read and deny-write modes (`ioDenyModes` set to $0021). Browsing access is traditional read-only access; it permits multiple readers but no writers. This access mode is useful for shared files that do not change often, such as help files, configuration files, and dictionaries.
- **Exclusive access.** You request exclusive access by specifying both read and write access and both deny-read and deny-write access (`ioDenyModes` set to $0033). Most applications that are not specifically designed to share file data use this permission setting. An exclusive access opening call succeeds only if there are no existing paths to the file. After a successful opening call, all future attempts to establish access paths to the file are denied until the exclusive-access path is closed.
- **Access as a single writer with multiple readers.** You request access as the single writer with multiple readers by specifying both read and write access and deny-write access (`ioDenyModes` set to $0023). This access method allows additional users to gain read-only access to browse a document being modified by the initial writer. The writer's application is responsible for range locking the file (by calling `PBLockRange`) before writing to it, to prevent reading when the file is inconsistent.
- **Shared access.** You request shared access by specifying both read and write access (`ioDenyModes` set to $0003). Shared access should be used by applications that support full multi-user access to its documents. Range locking is needed to prevent other users from accessing information undergoing change. Each user must also check for and handle any errors that result from access by other users. You might prefer to use a semaphore to flag records in the document as they are checked out, rather than use range locking exclusively.

You can open a shared file using either the deny modes described here or the file access permissions described in "File Manipulation" on page 2-7. If you use the original permissions when you open a file located in a shared directory, the File Manager translates those permissions into the corresponding access and deny modes. The basic rule followed in this translation is to allow a single writer or multiple readers, but not both. The translation from the original permissions to the deny-mode permissions is shown in Table 2-9.

Table 2-9 Access mode translation

HFS permissions	Deny-mode permissions
`fsCurPerm`	Exclusive access, or browsing access if exclusive access is unavailable.
`fsRdPerm`	Browsing access.
`fsWrPerm`	Exclusive access.
`fsRdWrPerm`	Exclusive access, or browsing access if exclusive access is unavailable.
`fsRdWrShPerm`	Shared access.

Notice that `fsCurPerm` and `fsRdWrPerm` are retried as read-only (browsing access) if exclusive access is not available. In addition, whenever browsing access is requested (that is, when you directly request `fsRdPerm`, or when a request for `fsCurPerm` or `fsRdWrPerm` is retried because exclusive access is not available) and cannot be granted, the AppleShare external file system searches through the open file control blocks (FCBs) for another AFP access path to the file. If an AFP access path to that file is found, a read-only access path is returned that shares the AFP access path.

Directory Access Privileges

AppleShare allows users to assign **directory access privileges** to individual directories, controlling who has access to the files and folders in the directory. A directory may be kept private, shared by a group of registered users, or shared with all users on the network.

Users are organized into groups. Users can belong to more than one group. Information about users and their privileges is maintained by AppleShare. Each directory has access privileges assigned for each of these three classifications of users: owner, group, and everyone. The following privileges can be assigned:

- **See Folders.** A user with this access privilege (also called **search privilege**) can see other directories in the specified directory.
- **See Files.** A user with this access privilege (also called **read privilege**) can see the icons and open documents or applications in that directory as well.
- **Make Changes.** A user with this access privilege (also called **write privilege**) can create, modify, rename, or delete any file or directory contained in the specified directory. Directory deletion requires additional privileges. It is possible to have Make Changes privileges without also having See Folders or See Files privileges; this would allow users to put items into a directory but not view the contents of that directory.

For instance, a user might assign privileges to a particular directory allowing the owner to read, write, and search the directory, and allowing everyone else (whether in the group or not) only to search the directory.

On directories shared using File Sharing, you can also assign **blank access privileges.** In this case, the File Manager ignores any other access privileges and uses the access privileges of the directory's parent. On the local machine, directories in a shared area have blank access privileges, until set otherwise.

Note

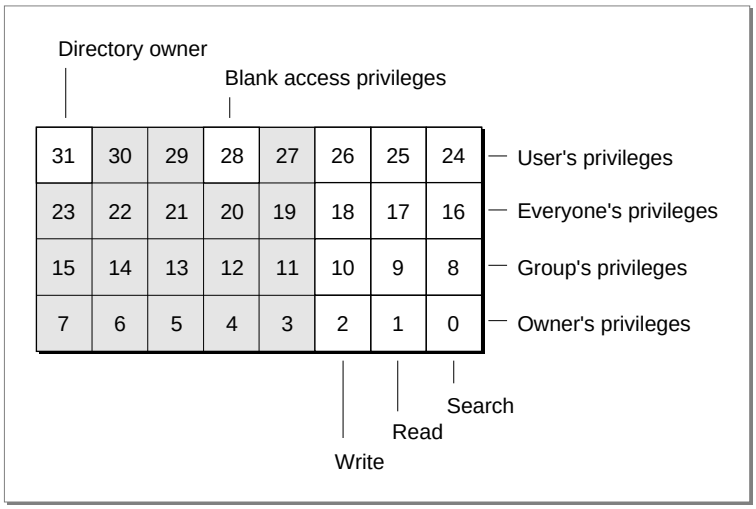
You cannot assign blank access privileges to a volume's root directory. ◆

You can use the `PBHGetDirAccess` and `PBHSetDirAccess` functions to determine and change the access privileges for a directory. The access privileges are passed in the 4-byte `ioACAccess` field of the `accessParam` variant of the HFS parameter block passed to these two functions. The 4 bytes are interpreted separately; byte 0 is the high-order byte.

Byte	Meaning
0	User's access privileges
1	Everyone's access privileges
2	Group's access privileges
3	Owner's access privileges

The bits in each byte encode access privilege information, as illustrated in Figure 2-2. (The high-order byte is on top, and the high-order bit is on the left.) Note that the user's privileges byte also indicates whether the user owns the directory and whether the directory has blank access privileges.

Figure 2-2 Access privileges information in the `ioACAccess` field



If bit 31 is set, then the user is the owner of the specified directory. If bit 28 is set, the specified directory has blank access privileges. If bit 28 is clear, the 3 low-order bits of each byte encode the write, read, and search privileges, respectively. If one of these bits is set, the directory privileges permit the indicated access to the specified individual.

The 3 low-order bits of the byte encoding the user's access privilege information are the logical `OR` of the corresponding bits in whichever of the other 3 bytes apply to the user. For example, if the user is the owner of a directory and is in the directory's group, then the 3 low-order bits of the user byte are the logical `OR` of the corresponding bits in the other 3 bytes. If, however, the user is not the owner and is not in the directory's group, the user privilege bits have the same values as the corresponding ones in the everyone byte.

You can use `PBHSetDirAccess` to set the low-order 3 bits of all the privileges bytes except the user's privileges byte. In the user's privileges byte, you can set only the blank access privileges bit (bit 28).

Note
Not all volumes support blank access privileges. You can call the `PBHGetVolParms` function to determine whether a particular volume supports blank access privileges. ◆

Remote Volume Mounting

Typically, the user mounts remote shared volumes through the Chooser or by opening an alias file. The File Manager in system software version 7.0 and later provides a set of calls for collecting the mounting information from a mounted volume and then using that information to mount the volume again later, without going through the Chooser.

Ordinarily, before you can mount a volume programmatically, you must record its mounting information while it's mounted. Because the size of the mounting information can vary, you first call the `PBGetVolMountInfoSize` function, which returns the size of the record you'll need to allocate to hold the mounting information. You then allocate the record and call `PBGetVolMountInfo`, passing a pointer to the record. When you want to mount the volume later, you can pass the record directly to the `PBVolumeMount` function.

Note
The functions for mounting volumes programmatically are low-level functions designed for specialized applications. Even if your application needs to track and access volumes automatically, it can ordinarily use the Alias Manager, described in the chapter "Alias Manager" in this book. The Alias Manager can record mounting information and later remount most volumes, even those that do not support the programmatic mounting functions. ◆

The programmatic mounting functions can now be used to mount AppleShare volumes. The functions have been designed so that they can eventually be used to mount local Macintosh volumes, such as partitions on devices that support partitioning, and local or remote volumes managed by non-Macintosh file systems.

Privilege Information in Foreign File Systems

Virtually every file system has its own **privilege model,** that is, conventions for controlling access to stored files and directories. A number of non-Macintosh file systems support access from a Macintosh computer by mapping their native privilege models onto the model defined by the **AppleTalk Filing Protocol (AFP).** Most applications that manipulate files in foreign file systems can rely on the intervening software to translate AFP privileges into whatever is required by the remote system.

The correlation is not always simple, however, and some applications require more control over the files stored on the foreign system. The A/UX privilege model, for example, recognizes four kinds of access: read, write, execute, and search. The AFP model recognizes read, write, deny-read, and deny-write access. If a shell program running on the Macintosh Operating System wants to allow the user to set native A/UX privileges on a remote file, it has to communicate with the A/UX file system using the A/UX privilege model.

System software version 7.0 provides two new functions, `PBGetForeignPrivs` and `PBSetForeignPrivs`, for manipulating privileges in a non-Macintosh file system. These access-control functions were designed for use by shell programs, such as the Finder, that need to use the native privilege model of the foreign file system. Most applications can rely on using shared environment functions, which are recognized by file systems that support the Macintosh privilege model. The new access-control functions do not relieve a foreign file system of the need to map its own privilege model onto the shared environment functions.

Like all other low-level File Manager functions, the access-control functions exchange information with your application through parameter blocks. The meanings of some fields vary according to the foreign file system used. These fields are currently defined for A/UX, and you can define them for other file systems.

You can identify the foreign file system through the `PBHGetVolParms` function. The attributes buffer introduced in system software version 7.0 for the `PBHGetVolParms` function contains a field for the foreign privilege model, `vMForeignPrivID`.

Note
The value of `vMForeignPrivID` does not specify whether the remote volume supports the AFP access-control functions. You can determine whether the volume supports the AFP access-control functions by checking the `bAccessCntl` bit in the `vMAttrib` field. ◆

A value of 0 for `vMForeignPrivID` signifies an HFS volume that supports no foreign privilege models. The field currently has one other defined value.

```
CONST
   fsUnixPriv = 1;            {A/UX privilege model}
```

For an updated list of supported models and their constants and fields, contact Macintosh Developer Technical Support.

A volume can support no more than one foreign privilege model.

The access-control functions store information in an HFS parameter block of type `foreignPrivParam`. The parameter block can store access-control information in one or both of

- a buffer of any length, whose location and size are stored in the parameter block
- 4 long words of data stored in the parameter block itself

The meanings of the fields in the parameter block depend on the definitions established by the foreign file system. For example, the A/UX operating system uses the `ioForeignPrivBuffer` field to point to a 16-byte buffer that describes the access rights for the specified file or directory. The buffer is divided into four fields, as follows:

Bytes	Description
0–3	The user ID of the owner of the file or directory.
4–7	The group ID of the owner of the file or directory.
8–11	Mode bits specifying the type of access available to the owner of the file or directory, the group of the file or directory, and to everyone else. The value in this field is a logical `OR` of some of the following octal values:

Value	Meaning
0001	Executable by others.
0002	Writable by others.
0004	Readable by others.
0010	Executable by the group.
0020	Writable by the group.
0040	Readable by the group.
0100	Executable by the owner.
0200	Writable by the owner.
0400	Readable by the owner.
2000	Set group ID on execution.
4000	Set user ID on execution.

(Execute privileges on a directory mean that the directory is searchable.) You can also use these octal masks to test or set common acess rights:

Mask	Meaning
0007	Executable, writable, and readable by others.
0070	Executable, writable, and readable by the group.
0700	Executable, writable, and readable by the owner.

12–15 The active user's access rights. The value in this field is a logical `OR` of some of the following octal values:

Value	Meaning
0001	Executable by user.
0002	Writable by user.
0004	Readable by user.
0010	Set if user owns this file or directory.

Note that you cannot change the owner of a file or directory using `PBSetForeignPrivs`. Accordingly, the value 0010 is meaningful for `PBGetForeignPrivs` only.

File ID Reference Routines

The File Manager provides a set of three low-level functions for creating, resolving, and deleting file ID references. These functions were developed for use by the Alias Manager in tracking files that have been moved within a volume or renamed. In most cases, you should use the Alias Manager, not file IDs, to track files. See the chapter "Alias Manager" in this book.

You establish a file ID reference when you need to identify a file using a file number (see "File IDs" on page 2-24). You create a file ID reference with the `PBCreateFileIDRef` function. Because the File Manager assigns file numbers independently on each volume, file IDs are not unique across volumes.

You can resolve a file ID reference by calling the `PBResolveFileIDRef` function, which determines the name and parent directory ID of the file with a given ID. If you no longer need a file ID, remove its record from the directory by calling the `PBDeleteFileIDRef` function.

Note
Removing a file ID is seldom appropriate, but the function is provided for completeness. ◆

Identifying Files, Directories, and Volumes

Whenever you want to perform some operation on a file, directory, or volume, you need to identify the target item to the File Manager. Exactly how you specify these items in the file system depends on several factors, including which version of system software is currently running and, if the target item is a file, whether it is open or closed. For example, once you have opened a file, you subsequently identify that file to the File Manager by providing its **file reference number,** a unique number returned to your application when you open the file.

In all other cases, you can identify files, directories, and volumes to the File Manager by using a variety of methods. In addition to file reference numbers, the File Manager recognizes

- file system specifications
- file ID references
- directory ID numbers
- volume reference numbers
- working directory reference numbers
- names and full or partial pathnames

This section describes each of these ways to identify items in the file system. Note, however, that some of these methods are of historical or theoretical interest only. Working directory reference numbers exist solely to provide compatibility with the

now-obsolete **Macintosh file system (MFS),** and their use is no longer recommended. Similarly, the use of full pathnames to specify volumes, directories, or files is not generally recommended.

Whenever possible, you should use file system specifications to identify files and directories because they provide the simplest method of identification and are recognized by the Finder, the Standard File Package, and other system software components beginning with system software version 7.0. If your application is intended to run in system software versions in which the routines that accept file system specification records are not available, you should use the volume reference number, parent directory ID, and name of the item you wish to identify.

File System Specifications

Conventions for identifying files, directories, and volumes have evolved as the File Manager has matured. System software version 7.0 introduced a simple, standard form for identifying a file or directory, called a **file system specification.** You can use a file system specification whenever you must identify a file or directory for the File Manager.

A file system specification contains

- the volume reference number of the volume on which the file or directory resides
- the directory ID of the parent directory
- the name of the file or directory

For a complete description of the file system specification (`FSSpec`) record, see “File System Specification Record” on page 2-87.

The Standard File Package in system software version 7.0 uses `FSSpec` records to identify files to be saved or opened. The File Manager provides a new set of high-level routines that accept `FSSpec` records as input, so that your application can pass the data directly from the Standard File Package to the File Manager. The Alias Manager and the Edition Manager accept file specifications only in the form of `FSSpec` records.

The Finder introduced in version 7.0 uses alias records, which are resolved into `FSSpec` records, to identify files to be opened or printed.

Version 7.0 also introduced the `FSMakeFSSpec` function, which initializes an `FSSpec` record for a particular file or directory. For a description of `FSMakeFSSpec`, see “Creating File System Specification Records” on page 2-35.

File IDs

A **file ID** is a unique number that the File Manager assigns to a file at the time it is created. The File Manager uses file IDs to distinguish one file from another on the same volume. In fact, a file ID is simply the catalog node ID of a file. As a result, file IDs are functionally analogous to directory IDs (described in the next section), and both kinds of IDs are assigned from the same set of numbers.

The File Manager can set up an internal record in the volume's catalog that specifies the filename and parent directory ID of the file with a given file ID, allowing you to reference the file by that number. (For more information about the volume's catalog, see "Catalog Files" on page 2-71.) This internal record in the volume catalog is a **file ID reference** (or **file ID thread record**).

It is important to distinguish file IDs from file ID references. File IDs exist on all HFS volumes, but file ID references might or might not exist on a particular HFS volume. Even if file ID references do exist on a volume, they might not exist for all the files on that volume. In addition, you can track files by their file IDs only on systems capable of creating and resolving file ID references. See "File ID Reference Routines" on page 2-23 for a description of the File Manager functions that allow you to manipulate file IDs.

Note

The file ID is a low-level tool and is unique only on one HFS volume. In most cases, your application should track files using the Alias Manager, described in the chapter "Alias Manager" in this book. The Alias Manager can track files across volumes. It creates a detailed record describing a file that you want to track, and, when you need to resolve the record later, it performs a sophisticated search. The Alias Manager uses file IDs internally. ◆

A file ID is analogous to a directory ID. A file ID is unique only within a volume and remains constant even when the file is moved or renamed. When a file is copied or restored from backup, however, the file ID changes. File IDs are unique over time—that is, once an ID has been assigned to a file, that number is not reused even after the file has been deleted.

The file ID is a permanent file reference, one that a user cannot change. After storing a file ID, your application can locate a specific file quickly and automatically, even if the user has moved or renamed it on the same volume.

File IDs are intended only as a tool for tracking files, not as a new element in file specification conventions. Neither high-level nor low-level File Manager functions accept file IDs as parameters.

Directory IDs

A **directory ID** is a unique number that the File Manager uses to distinguish one directory from another on the same volume. Assigned by the File Manager when the directory is created, a directory ID is simply the catalog node ID of a directory. As a result, directory IDs are functionally equivalent to file IDs, and both kinds of IDs are assigned from the same set of numbers.

Directory IDs are long integers. The File Manager defines several constants to refer to special directory IDs that exist on every volume.

```
CONST
   fsRtParID   =  1;    {directory ID of root directory's parent}
   fsRtDirID   =  2;    {directory ID of volume's root directory}
```

The root directory of every volume has a directory ID of 2. In addition, the root directory of every volume has a parent directory ID of 1. There is, however, no such parent directory; the constant `fsRtParID` is provided solely for use by applications and File Manager routines that need to specify a parent ID when referring to the volume's root directory. For example, if you call the `PBGetCatInfo` function when the `ioDirID` field is set to `fsRtDirID`, the value `fsRtParID` is returned in the `ioDrParID` field.

Volume Reference Numbers

A **volume reference number** is a unique number assigned to a volume at the time it is mounted. Unlike the volume name (which the user can change at any time and hence may not be unique), the volume reference number is both unique and unchangeable by the user, and so is a reliable way to refer to a volume for as long as it is mounted.

Volume reference numbers are small negative integers. They are valid only until the volume is unmounted. For example, if you place a volume offline and then bring it back online, that volume retains the same volume reference number it was originally assigned. However, if you unmount a volume and then remount it at some later time, its volume reference number might not be the same during both mounts.

Note
A volume reference number refers to a volume only as long as the volume is mounted. To create a volume reference that remains valid across subsequent boots, use alias records. See the chapter "Alias Manager" in this book for details. ◆

Working Directory Reference Numbers

The File Manager provides a method of identifying directories known as working directory reference numbers. A **working directory** is a temporary directory reference that the File Manager uses to specify both a directory and the volume on which it resides. Each working directory is assigned a **working directory reference number** at the time it is created. You can use this number in place of a volume reference number in all File Manager routines.

Note
Working directories were developed to allow applications written for the now-obsolete Macintosh file system to execute correctly when accessing volumes using the hierarchical file system. In general, your application should not create working directories and, in the few instances a working directory reference number is returned to your application, it should immediately convert that number to a volume reference number and directory ID. ◆

The first file system available on Macintosh computers was the **Macintosh file system (MFS)**, a "flat" file system in which all files are stored in a single directory. The hierarchical organization of folders within folders is an illusion maintained by the system software. As a result, you can identify a file under MFS simply by specifying its name and its volume. Typically, MFS routines require a volume reference number and a filename to specify a file.

To improve performance, especially with larger volumes, Apple Computer, Inc., introduced the **hierarchical file system (HFS)** on the Macintosh Plus computer and later models. In HFS, a volume can be divided into smaller units known as directories, which can themselves contain files or other directories. This hierarchical relationship of folders corresponds to an actual hierarchical directory structure maintained on disk. (See "Data Organization on Volumes" beginning on page 2-53 for the precise details of this hierarchical directory structure.)

Each file on an HFS volume is stored in a directory, called the file's **parent directory.** To identify a file in HFS, you must specify its volume, its parent directory, and its name. The File Manager assigns each directory a directory ID, and the user or the system software assigns each directory a name. The HFS File Manager routines include an additional parameter to handle the directory specification.

To keep existing applications running smoothly, Apple Computer, Inc. introduced the concept of working directories. A working directory is a combined directory and volume specification. To make a directory into a working directory, the File Manager establishes a **working directory control block** that contains both the volume and the directory ID of the target directory. The File Manager returns a unique working directory reference number, which you can use instead of the volume reference number in all routines.

Note
If your application provides both a directory ID and a working directory reference number, the directory ID is used to specify the directory (overriding the working directory specified by the working directory reference number). The working directory reference number is used to specify the volume (unless a volume name, which overrides all other forms of volume specification, is also provided). ◆

The best course of action is to avoid using working directories altogether. In the few cases where system software returns a working directory reference number to your application, the recommended practice is to immediately convert that working directory reference number into its corresponding directory ID and volume reference number (using `PBGetWDInfo` or its high-level equivalent, `GetWDInfo`).

In system software versions 7.0 and later, the Process Manager closes all working directories opened on behalf of your application when it terminates (quits or crashes). If your application might also run under earlier system software versions, you need to be careful to close any such working directories before you quit (using `PBCloseWD` or its high-level equivalent, `CloseWD`).

Names and Pathnames

Volumes, directories, and files all have names. A **volume name** is any sequence of 1 to 27 characters, excluding colons (:), that is assigned to a volume. File and directory names consist of any sequence of 1 to 31 characters, excluding colons. You can use uppercase and lowercase letters in names, but the File Manager ignores case when comparing names. The File Manager does not, however, ignore diacritical marks when comparing names.

Note
Although it is legal to use any character other than the colon in file, directory, and volume names, you should avoid using nonprinting characters in such names, even for temporary files that do not appear on the desktop or in the Standard File Package dialog boxes. A program written in C interprets a null character (ASCII code $00) as the end of a name; as a result, embedding the null character in a filename is likely to cause problems. In addition, file, directory, or volume names with null characters are not usable by AFP file servers (such as computers running Macintosh File Sharing or AppleShare software). In general, you should ensure that you use only printing characters in names of objects that you create in the file system. ◆

Files and directories located in the same directory must all have unique names. However, there is no requirement that volumes have unique names. It is perfectly acceptable for two mounted volumes to have the same name. This is one reason why your application should use volume reference numbers rather than volume names to specify volumes.

You can also specify files and directories using **pathnames,** although this method is discouraged. There are two kinds of pathnames, full and partial. A **full pathname** is a sequence of directory names, separated by colons, starting from the root directory (or volume) and leading down to the file. A full pathname to the file "Bananas," for instance, might be something like this:

```
MyVolume:Fruits:Tropical:Bananas
```

A **partial pathname** is a pathname that begins in some directory other than the root directory. A particular directory is specified by volume reference number (in the case of the root directory), working directory reference number, or directory ID, and the pathname begins relative to that directory. If the directory "Fruits" were specified, for instance, the partial pathname to the "Bananas" file would be

```
:Tropical:Bananas
```

The use of pathnames, however, is highly discouraged. If the user changes names or moves things around, they are worthless. It's best to stay with simple file or directory names and specify the directory containing the file or directory by its directory ID.

HFS Specifications

The simplest way to identify a mounted volume is by giving its volume reference number. The simplest way to identify a file or directory located on a mounted volume is by providing a file system specification. In some cases, however, you might not be able to use file system specifications.

For example, the low-level File Manager routines do not accept file system specifications, and so you must specify files and directories by some other method. You must also use another file-identification method when you use the high-level HFS routines that existed prior to the introduction of the routines that accept FSSpec records as file or directory

specifications. This section summarizes the conventions the File Manager uses to interpret the various volume, directory, and file specifications that are available even when file system specifications are not.

The File Manager recognizes three kinds of file system objects: files, directories, and volumes. You can identify them using various methods.

Object	Method of identification
File	Filename
Directory	Directory name
	Directory ID
	Working directory reference number, which also implies a volume
Volume	Volume name
	Volume reference number
	Working directory reference number, which also implies a directory

In HFS, you can pass a complete file specification in any of several ways:

- full pathname
- volume reference number and partial pathname
- working directory reference number and partial pathname
- volume reference number, directory ID, and partial pathname

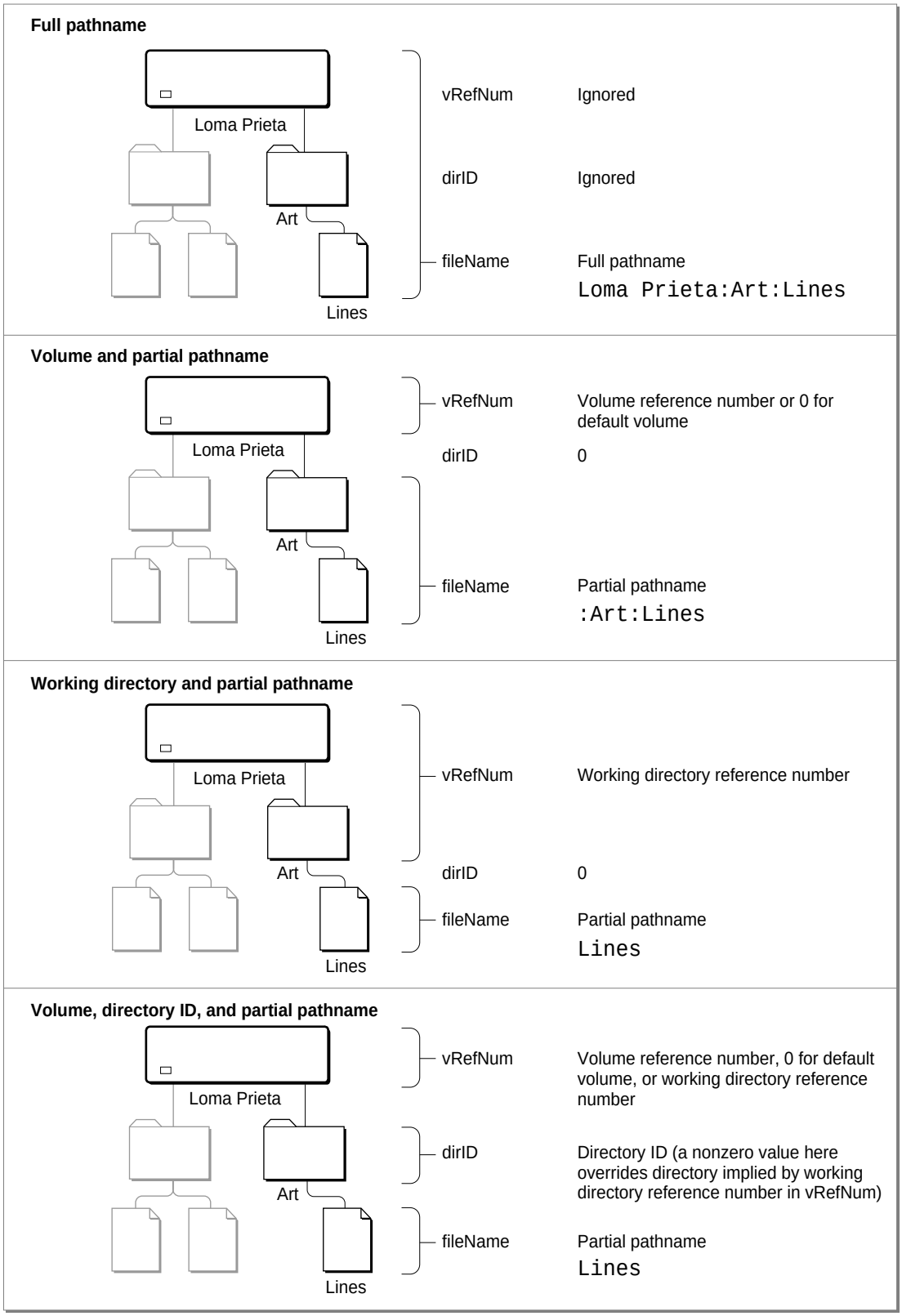
A full pathname consists of the name of the volume, the names of all directories between the root directory and the target, and the name of the target. A full pathname starts with a character other than a colon and contains at least one colon. If the first character is a colon, or if the pathname contains no colons, it is a partial pathname. If a partial pathname starts with the name of a parent directory, the first character in the pathname must be a colon. If a partial pathname contains only the name of the target file or directory, the leading colon is optional.

You can identify a volume in the `vRefNum` parameter by volume reference number or drive number, but volume reference number is preferred. A value of 0 represents the default volume. A volume name in the pathname overrides any other volume specification. Unlike a volume name, a volume reference number is guaranteed to be unique. It changes, however, each time a volume is mounted.

A working directory reference number represents both the directory ID and the volume reference number. If you specify any value other than 0 for the `dirID` parameter, that value overrides the directory ID implied by a working directory reference number in the volume parameter. The volume specification remains valid.

Figure 2-3 illustrates the standard ways to identify a file in HFS.

Figure 2-3 Identifying a file in HFS



Search Paths

Whenever you specify a value of 0 for the directory ID in an HFS specification, the File Manager first looks for the desired file in the directory indicated by the two other relevant HFS parameters or fields—namely, the pathname and the volume specification. If the specified file is not found in that directory, the File Manager continues searching for the file along a path known as the **poor man's search path.** You need to be aware of this behavior so that you do not accidentally open, delete, or otherwise manipulate the wrong file.

Note
The File Manager uses the poor man's search path only when the directory ID parameter or field has the value 0. You can avoid the consequences of accidentally opening or deleting the wrong file by specifying a directory explicitly with its directory ID. ◆

If the volume specification is a working directory reference number, the File Manager searches in the directory whose directory ID is encoded in that working directory reference number. If the volume specification is a volume reference number or 0, the File Manager searches in the default directory on the indicated volume. (See "Manipulating the Default Volume and Directory" on page 2-36 for information about default directories.) If you provide a full pathname, the File Manager searches in the directory whose name is contained in the pathname.

If the File Manager cannot find the specified file in the first directory it searches, it next searches the root directory of the boot volume, but only if the first directory searched is located on the boot volume. If the specified file is still not found, or if the first directory searched is not located on the boot volume, the File Manager next searches the System Folder, if one exists, on the volume containing the first directory searched. If the file still cannot be found, the File Manager gives up and returns the result code `fnfErr` (file not found) to your application.

As you can see, the use of the poor man's search path might lead to unexpected results. Suppose, for example, that you call the `HOpenDF` function like this:

```
myErr := HOpenDF(0, 0, ':Ackees', fsRdWrPerm, myRefNum);
```

The values of 0 for the first two parameters (the volume specification and directory ID) indicate that you want the File Manager to look for the named file in the default directory. If, however, there is no such file in that directory, the File Manager continues looking along the poor man's search path for a file with the specified name. The result might be that you open the wrong file. (Worse yet, if you had called `HDelete` instead of `HOpenDF`, you might have deleted the wrong file!)

The File Manager uses the poor man's search path for all routines that can return the `fnfErr` result code and to which you passed a directory ID of zero. It does not use the poor man's search path when you specify a nonzero directory ID or when you call an indexed routine (that is, when the `ioFDirIndex` field of the parameter block has a nonzero value). The File Manager also does not use the poor man's search path when you create a file (perhaps by calling `PBHCreate`) or move a file between directories (by calling `PBCatMove`).

Note
The poor man's search path might not be supported in future versions of system software. You should not depend on its availability. ◆

Using the File Manager

You can use the File Manager to manipulate files, directories, and volumes. The chapter "Introduction to File Management" in this book shows how to use the File Manager and other system software services to accomplish the most common file-related operations (that is, handling the typical File menu commands). This section shows how to accomplish a variety of other operations on files, directories, and volumes. In particular, this section shows how to

- determine the available features of the File Manager
- determine the characteristics of a particular mounted volume
- create file system specification records
- manipulate the default volume and directory
- delete files and file forks
- search a volume for files or directories matching various criteria
- construct the full pathname of a file
- determine the amount of free space on a volume
- lock and unlock byte ranges in shared files

Altogether, the code listings given in this section provide a rich source of information about using the many File Manager routines and data structures.

Determining the Features of the File Manager

Some of the capabilities provided by the File Manager depend on the version of system software that is running, and some others depend on the characteristics of the target volume. For example, the routines that accept `FSSpec` records as file or directory specifications were introduced in system software version 7.0 and are unavailable in earlier system software versions—unless your software development system provides "glue" that allows you to call those routines when running in earlier system software versions (or unless some system extension provides those routines). Similarly, some volumes support features that other volumes do not; a volume that has local file sharing enabled, for instance, allows you to lock byte ranges in any files on a volume that is sharable.

Before using any of the File Manager features that are not universally available in all system software versions and on all volumes, you should check for that feature's availability by calling either the `Gestalt` function or the `PBHGetVolParms` function, according to whether the feature's presence depends on the system software or the characteristics of the volume.

You can use Gestalt to determine whether or not you can call the functions that accept and support FSSpec records. Call Gestalt with the gestaltFSAttr selector to check for File Manager features. The response parameter currently has two relevant bits:

```
CONST
   gestaltFullExtFSDispatching   =  0;    {exports HFSDispatch traps}
   gestaltHasFSSpecCalls         =  1;    {supports FSSpec records}
```

Constant descriptions

gestaltFullExtFSDispatching
: If set, all of the routines selected through the _HFSDispatch trap are available to external file systems. If this bit is clear, the File Manager checks the selector passed to _HFSDispatch and ensures that it is valid; if the selector is invalid, the result code paramErr is returned to the caller. If this bit is set, no such validity checking is performed.

gestaltHasFSSpecCalls
: If set, the operating environment provides the file system specification versions of the basic file-manipulation functions, plus the FSMakeFSSpec function.

The chapter "Introduction to File Management" in this book illustrates how to use the Gestalt function to determine whether the operating environment supports the routines that accept FSSpec records. For a complete description of the Gestalt function, see the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

To test for the availability of the features that depend on the volume, you can call the low-level function PBHGetVolParms. Listing 2-1 illustrates how you can determine whether the PBCatSearch function is available before using it to search a volume's catalog. Note that the SupportsCatSearch function defined in Listing 2-1 first calls Gestalt to determine whether the File Manager supports PBCatSearch. If it does, the SupportsCatSearch function calls PBHGetVolParms to see if the indicated volume also supports PBCatSearch.

Listing 2-1 Testing for PBCatSearch

```
FUNCTION SupportsCatSearch (vRefNum: Integer): Boolean;
VAR
   myHPB:         HParamBlockRec;
   infoBuffer:    GetVolParmsInfoBuffer;
   attrib:        LongInt;
BEGIN
   SupportsCatSearch := FALSE;   {assume no PBCatSearch support}
   IF gHasGestalt THEN           {set this somewhere else}
      IF Gestalt(gestaltFSAttr, attrib) = noErr THEN
         IF BTst(attrib, gestaltFullExtFSDispatching) THEN
```

```
            BEGIN                {this File Mgr has PBCatSearch}
               WITH myHPB DO
                  BEGIN
                     ioNamePtr := NIL;
                     ioVRefNum := vRefNum;
                     ioBuffer := @infoBuffer;
                     ioReqCount := SIZEOF(infoBuffer);
                  END;
               IF PBHGetVolParms(@myHPB, FALSE) = noErr THEN
                  IF BTST(infoBuffer.vMAttrib, bHasCatSearch) THEN
                     SupportsCatSearch := TRUE;
            END;
END;
```

The `SupportsCatSearch` function calls `PBHGetVolParms` for the volume whose reference number is passed as a parameter to `SupportsCatSearch`. The `PBHGetVolParms` function returns information about a volume in a record of type `GetVolParmsInfoBuffer`. The `vMAttrib` field of that record contains a number of bits that encode information about the capabilities of the target volume. In particular, the bit `bHasCatSearch` is set if the specified volume supports the `PBCatSearch` function.

Note
Some features of volumes might change dynamically during the execution of your application. For example, the user can turn File Sharing on and off, thereby changing the capabilities of volumes. See "Locking and Unlocking File Ranges" on page 2-51 for more details. ◆

Creating File System Specification Records

Sometimes it is useful for your application to create a file system specification record. For example, your application might be running in an environment where the enhanced Standard File Package routines (which return `FSSpec` records) are unavailable but the File Manager routines that accept `FSSpec` records are available (perhaps as glue code in your development system). You can call the `FSMakeFSSpec` function (or its low-level equivalent `PBMakeFSSpec`) to initialize a file system specification record.

Three of the parameters to `FSMakeFSSpec` represent the volume, parent directory, and file specifications of the target object. You can provide this information in any of the four combinations described in "HFS Specifications" beginning on page 2-28. Table 2-10 details the ways your application can identify the name and location of a file or directory in a call to `FSMakeFSSpec`.

The fourth parameter to `FSMakeFSSpec` is a pointer to the `FSSpec` record.

Table 2-10 How FSMakeFSSpec interprets its parameters

vRefNum	dirID	fileName	Interpretation
Ignored	Ignored	Full pathname	Full pathname overrides any other information
Volume reference number or drive number	Directory ID	Partial pathname	Partial pathname starts in the directory whose parent is specified in the dirID parameter
Working directory reference number	Directory ID	Partial pathname	Directory specification in the dirID parameter overrides the directory implied by the reference number
			Partial pathname starts in the directory whose parent is specified in dirID
Volume reference number or drive number	0	Partial pathname	Partial pathname starts in the root directory of the volume in vRefNum
Working directory reference number	0	Partial pathname	Partial pathname starts in the directory specified by the working directory reference number
Volume reference number of drive	Directory ID	Empty string or NIL	The target object is the directory specified by the directory ID in dirID
Working directory reference number	0	Empty string or NIL	The target object is the directory specified by the working directory reference number in vRefNum
Volume reference number or drive number	0	Empty string or NIL	The target object is the volume specified in vRefNum
0	Directory ID	Empty string or NIL	The target object is the directory specified in dirID on the default volume
0	Directory ID	Partial pathname	Partial pathname starts in the directory specified in dirID on the default volume
0	0	Empty string or NIL	The target object is the default directory on the default volume
0	0	Partial pathname	Partial pathname starts in the default directory on the default volume

Manipulating the Default Volume and Directory

When your application is running, the File Manager maintains a default volume and a default directory for it. The **default directory** is used in File Manager routines whenever you don't explicitly specify some directory. The **default volume** is the volume containing the default directory.

If you pass 0 as the volume specification with routines that operate on a volume (such as mounting or ejecting routines), the File Manager assumes that you want to perform the operation on the default volume. Initially, the volume used to start up the application is set as the default volume, but your application can designate any mounted volume as the default volume.

With routines that access files or directories, if you don't specify a directory *and* you pass a volume specification of 0, the File Manager assumes that the file or directory is located in the default directory. Initially, the default directory is set to the root directory of the default volume, but your application can designate any directory as the default directory.

Note
Don't confuse the default directory and volume maintained by the File Manager with the current directory and volume maintained by the Standard File Package. Although the default volume and current volume are initially the same, they can differ whenever your application resets one of them. See the chapter "Standard File Package" in this book for more information about the current directory and volume. ◆

The provision of a default volume was originally intended as a convenient way for you to limit all File Manager calls to a particular volume. The default directory was introduced along with HFS as an analog to the default volume. In general, however, it is safest to specify both a volume and a directory explicitly in all File Manager calls. In particular, the introduction of file system specification records has rendered default volumes and directories largely obsolete. As a result, you should avoid relying on them.

In some cases, however, you might want to set the default volume or directory explicitly. You can determine the default volume and directory by calling the `GetVol` or `HGetVol` function. You can explicitly set the default directory and volume by calling the `SetVol` or `HSetVol` function. For reasons explained later, however, the use of `HSetVol` and its low-level equivalent `PBHSetVol` is discouraged.

To set the default volume only, you can call `SetVol`, passing it the volume reference number of the volume you want to establish as the default volume, as in this example:

```
myErr := SetVol(NIL, myVRefNum);
```

You can instead specify the volume by name, but because volume names might not be unique, it is best to use the volume reference number.

To set both the default directory and the default volume, you could call `HSetVol`, passing it the appropriate volume reference number and directory ID, as in this example:

```
myErr := HSetVol(NIL, myVRefNum, myDirID);
```

However, using `HSetVol` can lead to problems in certain circumstances. When you call `HSetVol` (or its low-level version `PBHSetVol`) and pass a working directory reference number in the `vRefNum` parameter, the File Manager stores the encoded volume reference number and directory ID separately. If you later call `GetVol` (or its low-level version `PBGetVol`), the File Manager returns that volume reference number, not the working directory reference number you passed to `HSetVol`. The net result is that any code using the results of the `GetVol` call will access the root directory of the default volume, not the actual default directory.

It is important to realize that calling `HSetVol` is perfectly safe if all the code executing in your application's partition always calls `HGetVol` instead of `GetVol`. This is because `HGetVol` returns a working directory reference number whenever the previous call to `HSetVol` passed one in. Calling `HSetVol` can create problems only if your application is running under a system software version prior to version 7.0. In that case, a desk accessory might be opened in your application's partition, thereby inheriting your application's default volume and directory. If that desk accessory calls `GetVol` instead of `HGetVol`, it might receive a volume reference number when it expects a working directory reference number, as described in the previous paragraph. To avoid this problem, you can simply use `SetVol` (or `PBSetVol`) instead of `HSetVol`, as in this example:

```
myErr := SetVol(NIL, myVRefNum);
```

In this case, the `myVRefNum` parameter should contain a working directory reference number.

Deleting Files and File Forks

You can delete a file by calling `FSpDelete`, `HDelete`, or `PBHDelete`. These functions delete both forks of a file by removing the catalog entry for the file and adjusting the volume information block and volume bitmap accordingly. These functions do not actually erase the disk areas occupied by the file, so there is a reasonable chance that a good disk utility might be able to salvage a deleted file if the user hasn't allocated any new file blocks in the meantime.

Sometimes you might want to truncate just one fork of a file. Listing 2-2 illustrates how you can truncate a file's resource fork while preserving the data fork.

Listing 2-2 Deleting a file's resource fork

```
FUNCTION TruncateRF (myFileSpec: FSSpec): OSErr;
VAR
   myErr:   OSErr;      {result code}
   myFile:  Integer;    {file reference number}
BEGIN
   myErr := FSpOpenRF(myFileSpec, fsRdWrPerm, myFile);
   IF myErr = noErr THEN
      myErr := SetEOF(myFile, 0);
   IF myErr = noErr THEN
      myErr := FSClose(myFile);
   IF myErr = noErr THEN
      myErr := FlushVol(myFileSpec.vRefNum);
   TruncateRF := myErr;
END;
```

The function TruncateRF defined in Listing 2-2 opens the file's resource fork with exclusive read / write permission and sets its logical end-of-file to 0. This effectively releases all the space occupied by the resource fork on the volume. Then TruncateRF closes the file and updates the volume.

Searching a Volume

To search a volume efficiently, you can use the PBCatSearch function. The PBCatSearch function looks at all entries in the volume's catalog file and returns a list of all files or directories that match the criteria you specify. You can ask PBCatSearch to match files or directories using many types of criteria, including

- names or partial names
- file and directory attributes
- Finder information
- physical and logical file length
- creation, modification, and backup dates
- parent directory ID

Like all low-level File Manager functions, PBCatSearch exchanges information with your application through a parameter block. The PBCatSearch function uses the csParam variant of the basic parameter block defined by the HParamBlockRec data type. That variant includes two fields, ioSearchInfo1 and ioSearchInfo2, that contain the addresses of two catalog information records (of type CInfoPBRec). You specify which kinds of files or directories you want to search for by filling in the fields of those two records.

The fields in ioSearchInfo1 and ioSearchInfo2 have different uses:

- The ioNamePtr field in ioSearchInfo1 holds a pointer to the target string; the ioNamePtr field in ioSearchInfo2 must be NIL. (If you're not searching for the name, the ioNamePtr field in ioSearchInfo1 must also be NIL.)
- The date and length fields in ioSearchInfo1 hold the lowest values in the target range, and the date and length fields in ioSearchInfo2 hold the highest values in the target range. The PBCatSearch function looks for values greater than or equal to the field values in ioSearchInfo1 and less than or equal to the values in ioSearchInfo2.
- The ioFlAttrib and ioFlFndrInfo fields in ioSearchInfo1 hold the target values, and the same fields in ioSearchInfo2 hold masks that specify which bits are relevant.

Some fields in the catalog information records apply only to files, some only to directories, and some to both. Some of the fields that apply to both have different names, depending on whether the target of the record is a file or a directory. The PBCatSearch function uses only some fields in the catalog information record. Table 2-11 lists the fields used for files.

Table 2-12 lists the fields in catalog information records used for directories.

Table 2-11 Fields in `ioSearchInfo1` and `ioSearchInfo2` used for a file

Field	Meaning in `ioSearchInfo1`	Meaning in `ioSearchInfo2`
`ioNamePtr`	Pointer to filename	Reserved (must be `NIL`)
`ioFlAttrib`	Desired file attributes	Mask for file attributes
`ioFlFndrInfo`	Desired Finder information	Mask for Finder information
`ioFlLgLen`	Smallest logical size of data fork	Largest logical size
`ioFlPyLen`	Smallest physical size of data fork	Largest physical size
`ioFlRLgLen`	Smallest logical size of resource fork	Largest logical size
`ioFlRPyLen`	Smallest physical size of resource fork	Largest physical size
`ioFlCrDat`	Earliest file creation date	Latest file creation date
`ioFlMdDat`	Earliest file modification date	Latest file modification date
`ioFlBkDat`	Earliest file backup date	Latest file backup date
`ioFlXFndrInfo`	Desired extended Finder information	Mask for Finder information
`ioFlParID`	Smallest directory ID of file's parent	Largest parent directory ID

Table 2-12 Fields in `ioSearchInfo1` and `ioSearchInfo2` used for a directory

Field	Meaning in `ioSearchInfo1`	Meaning in `ioSearchInfo2`
`ioNamePtr`	Pointer to directory name	Reserved (must be `NIL`)
`ioFlAttrib`	Desired directory attributes	Mask for directory attributes
`ioDrUsrWds`	Desired Finder information	Mask for Finder information
`ioDrNmFls`	Smallest number of files in directory	Largest number of files
`ioDrCrDat`	Earliest directory creation date	Latest creation date
`ioDrMdDat`	Earliest directory modification date	Latest modification date
`ioDrBkDat`	Earliest directory backup date	Latest backup date
`ioDrFndrInfo`	Desired extended Finder information	Mask for Finder information
`ioDrParID`	Smallest directory ID of directory's parent	Largest parent directory ID

The `PBCatSearch` function searches only on bits 0 and 4 in the file attributes field (`ioFlAttrib`).

Bit	Meaning
0	Set if the file or directory is locked.
4	Set if the item is a directory.

Note
The `PBCatSearch` function cannot use the additional bits returned in the `ioFlAttrib` field by the `PBGetCatInfo` function. ◆

To give PBCatSearch a full description of the search criteria, you pass it a pair of catalog information records that determine the limits of the search and a mask that identifies the relevant fields within the records. You pass the mask in the ioSearchBits field in the PBCatSearch parameter block. To determine the value of ioSearchBits, add the appropriate constants. To match all files and directories on a volume (including the volume's root directory), set ioSearchBits to 0.

```
CONST
   fsSBPartialName   =  1;    {substring of name}
   fsSBFullName      =  2;    {full name}
   fsSBFlAttrib      =  4;    {directory flag; software lock flag}
   fsSBNegate        =  16384;{reverse match status}
   {for files only}
   fsSBFlFndrInfo    =  8;    {Finder file info}
   fsSBFlLgLen       =  32;   {logical length of data fork}
   fsSBFlPyLen       =  64;   {physical length of data fork}
   fsSBFlRLgLen      =  128;  {logical length of resource fork}
   fsSBFlRPyLen      =  256;  {physical length of resource fork}
   fsSBFlCrDat       =  512;  {file creation date}
   fsSBFlMdDat       =  1024; {file modification date}
   fsSBFlBkDat       =  2048; {file backup date}
   fsSBFlXFndrInfo   =  4096; {more Finder file info}
   fsSBFlParID       =  8192; {file's parent ID}
   {for directories only}
   fsSBDrUsrWds      =  8;    {Finder directory info}
   fsSBDrNmFls       =  16;   {number of files in directory}
   fsSBDrCrDat       =  512;  {directory creation date}
   fsSBDrMdDat       =  1024; {directory modification date}
   fsSBDrBkDat       =  2048; {directory backup date}
   fsSBDrFndrInfo    =  4096; {more Finder directory info}
   fsSBDrParID       =  8192; {directory's parent ID}
```

For example, to search for a file that was created between two specified dates and whose name contains a specified string, set ioSearchBits to 517 (that is, to fsSBFlAttrib + fsSBFlCrDat + fsSBPartialName).

A catalog entry must meet all of the specified criteria to be placed in the list of matches. After PBCatSearch has completed its scan of each entry, it checks the fsSBNegate bit. If that bit is set, PBCatSearch reverses the entry's match status (that is, if the entry is a match but the fsSBNegate bit is set, the entry is not put in the list of matches; if it is not a match, it is put in the list).

Note

The fsSBNegate bit is ignored during searches of remote volumes that support AFP version 2.1. ◆

Although using PBCatSearch is significantly more efficient than searching the directories recursively, searching a large volume can still take long enough to affect user response time. You can break a search into several shorter searches by specifying a maximum length of time in the ioSearchTime field of the parameter block and keeping an index in the ioCatPosition field. The PBCatSearch function stores its directory-location index in a catalog position record, which is defined by the CatPositionRec data type.

```
TYPE CatPositionRec =                   {catalog position record}
RECORD
   initialize: LongInt;                 {starting point}
   priv:       ARRAY[1..6] OF Integer; {private data}
END;
```

To start a search at the beginning of the catalog, set the initialize field to 0. When PBCatSearch exits because of a timeout, it updates the record so that it describes the next entry to be searched. When you call PBCatSearch to resume the search after a timeout, pass the entire record that was returned by the last call. PBCatSearch returns a list of the names and parent directories of all files and directories that match the criteria you specify. It places the list in an array pointed to by the ioMatchPtr field.

Note

The ioSearchTime field is not used by AFP volumes. To break up a potentially lengthy search into smaller searches on AFP volumes, use the ioReqMatchCount field to specify the maximum number of matches to return. ◆

Listing 2-3 illustrates how to use PBCatSearch to find all files (not directories) whose names contain the string “Temp” and that were created within the past two days.

Listing 2-3 Searching a volume with PBCatSearch

```
CONST
   kMaxMatches       =  30;      {find up to 30 matches in one pass}
   kOptBufferSize    =  $4000;   {use a 16K search cache for speed}
VAR
   myErr:      OSErr;            {result code of function calls}
   myCount:    Integer;          {loop control variable}
   myFName:    Str255;           {name of string to look for}
   myVRefNum:  Integer;          {volume to search}
   myDirID:    LongInt;          {ignored directory ID for HGetVol}
   myCurrDate: LongInt;          {current date, in seconds}
   twoDaysAgo: LongInt;          {date two days ago, in seconds}
   myPB:       HParamBlockRec;   {parameter block for PBCatSearch}
   myMatches:  PACKED ARRAY[1..kMaxMatches] OF FSSpec;
                                 {put matches here}
```

```
   mySpec1:    CInfoPBRec;       {search criteria, part 1}
   mySpec2:    CInfoPBRec;       {search criteria, part 2}
   myBuffer:   PACKED ARRAY[1..kOptBufferSize] OF Char;
                                 {search cache}
   done:       Boolean;          {have all matches been found?}
PROCEDURE SetupForFirstTime;
BEGIN
   myErr := HGetVol(NIL, myVRefNum, myDirID);
                                          {search on the default volume}
   myFName := 'Temp';                     {search for "Temp"}
   GetDateTime(myCurrDate);               {get current time in seconds}
   twoDaysAgo := myCurrDate - (2 * 24 * 60 * 60);
   WITH myPB DO
   BEGIN
      ioCompletion   := NIL;              {no completion routine}
      ioNamePtr      := NIL;              {no volume name; use vRefNum}
      ioVRefNum      := myVRefNum;        {volume to search}
      ioMatchPtr     := FSSpecArrayPtr(@myMatches);
                                          {points to results buffer}
      ioReqMatchCount:= kMaxMatches;      {number of matches}
      ioSearchBits   := fsSBPartialName   {search on partial name}
                        + fsSBFlAttrib    {search on file attributes}
                        + fsSBFlCrDat;    {search on creation date}
      ioSearchInfo1  := @mySpec1;         {points to first criteria set}
      ioSearchInfo2  := @mySpec2;         {points to second criteria set}
      ioSearchTime   := 0;                {no timeout on searches}
      ioCatPosition.initialize := 0;      {set hint to 0}
      ioOptBuffer := @myBuffer;           {point to search cache}
      ioOptBufSize := kOptBufferSize;     {size of search cache}
   END;
   WITH mySpec1 DO
   BEGIN
      ioNamePtr := @myFName;              {point to string to find}
      ioFlAttrib := $00;                  {clear bit 4 to ask for files}
      ioFlCrDat := twoDaysAgo;            {lower bound of creation date}
   END;
   WITH mySpec2 DO
   BEGIN
      ioNamePtr := NIL;                   {set to NIL}
      ioFlAttrib := $10;                  {set mask for bit 4}
      ioFlCrDat := myCurrDate;            {upper bound of creation date}
   END;
END;
```

```
BEGIN
   SetupForFirstTime;                    {initialize data records}
   REPEAT
      myErr := PBCatSearchSync(@myPB);   {get some files}
      done := (myErr = eofErr);          {eofErr returned when all done}
      IF ((myErr = noErr) | done) & (myPB.ioActMatchCount > 0) THEN
         FOR myCount := 1 TO myPB.ioActMatchCount DO
            Writeln(myMatches[myCount].name);
                                         {report all matches found}
   UNTIL done;
END;
```

When PBCatSearch is not available in the current operating environment or is not supported by the volume you wish to search, you'll need to use PBGetCatInfo to perform a recursive, indexed search through the volume's directory hierarchy. This kind of search is usually much slower than a search with PBCatSearch, and you can encounter problems you avoid by using PBCatSearch. For example, a recursive, indexed search can require a large amount of stack space. The procedure EnumerateShell defined in Listing 2-4 is designed to minimize the amount of stack space used. As a result, it should execute even in environments with very limited stack space.

Listing 2-4 Searching a volume using a recursive, indexed search

```
PROCEDURE EnumerateShell (vRefNum: Integer; dirID: LongInt);
VAR
   myName:        Str63;
   myCPB:         CInfoPBRec;
   myErr:         OSErr;
   PROCEDURE EnumerateCatalog (dirID: LongInt);
   CONST
         kFolderBit = 4;
   VAR
         index:   Integer;
   BEGIN
      index := 1;
      REPEAT
         WITH myCBP DO
         BEGIN
            ioFDirIndex := index;
            ioDrDirID := dirID;     {reset dirID; PBGetCatInfo may change it}
            ioACUser := 0;
         END;
         myErr := PBGetCatInfo(@myCPB, FALSE);
         IF myErr = noErr THEN
```

```
            IF BTst(myCPB.ioFlAttrib, kFolderBit) THEN
               BEGIN {we have a directory}
                  {Do something useful with the dir. information in myCPB.}
                  EnumerateCatalog(myCPB.ioDrDirID);
                  myErr := noErr;      {clear error return on way back}
               END
            ELSE
               BEGIN {we have a file}
                  {Do something useful with the file information in myCPB.}
               END;
         index := index + 1;
      UNTIL (myErr <> noErr);
   END; {EnumerateCatalog}
BEGIN {EnumerateShell}
   WITH myCPB DO
      BEGIN
         ioNamePtr := @myName;
         ioVRefNum := vRefNum;
      END;
   EnumerateCatalog(dirID);
END; {EnumerateShell}
```

The `EnumerateShell` procedure sets up a catalog information parameter block with a pointer to a string variable and the volume reference number passed to it. It then calls the `EnumerateDir` procedure, which uses indexed calls to `PBGetCatInfo` to read the catalog information about all items in the specified directory. If an item is a directory (as indicated by the `kFolderBit` bit of the `ioFlAttrib` field of the parameter block), `EnumerateDir` calls itself recursively to enumerate the contents of that directory. If an item is a file, `EnumerateDir` performs whatever processing is appropriate.

Note that `EnumerateDir` resets the `ioDrDirID` field before calling `PBGetCatInfo`. This is necessary because `PBGetCatInfo` returns a file ID number in that field if the item is a file. The `EnumerateDir` procedure also clears the `ioACUser` field. You need to do this if your search depends on the value in that field after the call to `PBGetCatInfo`, because the value returned in that field for local volumes is meaningless.

To search an entire volume, call the `EnumerateShell` procedure with the `vRefNum` parameter set to the volume reference number of the volume you want to search and the `dirID` parameter set to `fsRtDirID`. You can also do a partial search of a volume by specifying a different directory ID in the `dirID` parameter.

Constructing Full Pathnames

As indicated in “Names and Pathnames” on page 2-27, the use of full or partial pathnames is strongly discouraged. Full pathnames are particularly unreliable as a means of identifying files or directories within your application, largely because the user can change the name of any element in the path at virtually any time. In general, you should use a file’s name, parent directory ID, and volume reference number to identify a file you want to open, delete, or otherwise manipulate.

If you need to remember the location of a particular file across subsequent system boots, use the Alias Manager to create an alias record describing the file. If the Alias Manager is not available, you can save the file's name, its parent directory ID, and the name of the volume on which it's located. Although none of these methods is foolproof, they are much more reliable than using full pathnames to identify files.

Nonetheless, it is sometimes useful to display a file's full pathname to the user. For example, a backup utility might display a list of full pathnames of files as it copies them onto the backup medium. Or, a utility might want to display a dialog box showing the full pathname of a file when it needs the user's confirmation to delete the file. No matter how unreliable full pathnames may be from a file-specification viewpoint, users understand them more readily than volume reference numbers or directory IDs.

Note

The following technique for constructing the full pathname of a file is intended for display purposes only. Applications that depend on any particular structure of a full pathname are likely to fail on alternate foreign file systems or under future system software versions. ◆

Listing 2-5 shows one way to define a function, `GetFullPath`, that accepts a directory ID and a filename as parameters and returns the full pathname of the corresponding file (if any). The `GetFullPath` function calls the low-level function `PBGetCatInfo` for the specified directory to determine the name and directory ID of that directory's parent directory. It then performs the same operation on the parent directory's parent, continuing until it finds a parent directory with ID `fsRtDirID`. Under HFS, this is always the ID of a volume's root directory.

Listing 2-5 Constructing the full pathname of a file

```
FUNCTION GetFullPath (DirID: LongInt; vRefnum: Integer): Str255;
VAR
   myPB:       CInfoPBRec;    {parameter block for PBGetCatInfo}
   dirName:    Str255;        {a directory name}
   fullPath:   Str255;        {full pathname being constructed}
   myErr:      OSErr;
BEGIN
   fullPath := '';               {initialize full pathname}
   myPB.ioNamePtr := @dirName;
   myPB.ioVRefNum := vRefNum;    {indicate target volume}
   myPB.ioDrParID := DirId;      {initialize parent directory ID}
   myPB.ioFDirIndex := -1;       {get info about a directory}
   {Get name of each parent directory, up to root directory.}
   REPEAT
      myPB.ioDrDirID := myPB.ioDrParID;
      myErr := PBGetCatInfo(@myPB, FALSE);
      IF gHaveAUX THEN
```

```
         BEGIN
            IF dirName[1] <> '/' THEN
               dirName := concat(dirName, '/');
         END
      ELSE
         dirName := concat(dirName, ':');
      fullPath := concat(dirName, fullPath);
   UNTIL myPB.ioDrDirID = fsRtDirID;
   GetFullPath := fullPath;      {return full pathname}
END;
```

Note that `GetFullPath` uses either a slash (/) or a colon (:) to separate names in the full path, depending on whether A/UX is running or not. The `GetFullPath` function reads the value of the global variable `gHaveAUX` to determine whether A/UX is running; your application must initialize this variable (preferably by calling the `Gestalt` function) before it calls `GetFullPath`.

The `GetFullPath` function defined in Listing 2-5 returns a result of type `Str255`, which limits the full pathname to 255 characters. An actual full pathname, however, might exceed 255 characters. A volume name can be up to 27 characters, and each directory name can be up to 31 characters. If the average volume and directory name is about 20 characters long, `GetFullPath` can handle files located only about 12 levels deep. If the length of the average directory name is closer to the maximum, `GetFullPath` provides a full pathname for files located only about 8 levels deep. If necessary, you can overcome this limitation by rewriting `GetFullPath` to return a handle to the full pathname; the algorithm for ascending the directory hierarchy using `PBGetCatInfo` will still work, however.

Determining the Amount of Free Space on a Volume

You can determine how much space is free on a particular volume by calling the low-level function `PBHGetVInfo`. This function returns, in the `ioVFrBlk` field of the parameter block passed to it, the number of free allocation blocks on a volume. It also returns, in the `ioVAlBlkSiz` field, the number of bytes in the allocation blocks on that volume. By multiplying those two values, you can determine how many bytes are free on a particular volume.

There is, however, one complication in this process. The `ioVFrBlk` field of the parameter block is actually an unsigned integer and can contain values from 0 to 65,535. However, because Pascal does not support unsigned integers, it interprets the values in the `ioVFrBlk` field as lying in the range –32,768 to 32,767. (Integers are stored as 16-bit quantities where the high-order bit indicates whether the value is true binary or a negated value in its two's complement positive form.) If, for example, a volume has 40,000 allocation blocks free and your application blindly returned the value in the `ioVFrBlk` field, it would erroneously report that the volume had –25,536 allocation blocks available.

You can circumvent this problem by forcing Pascal to interpret the high-order bit as part of the number of free blocks. For example, if you install the value returned in the

ioVFrBlk field as the low-order word of a long integer, the high-order bit of that word is no longer the high-order bit of that long integer and hence is not interpreted as a sign indication. The data type TwoIntsMakeALong provides a convenient way to accomplish this.

```
TYPE
   TwoIntsMakeALong =      {two integers make a long integer}
   RECORD
      CASE Integer OF
         1: (long: LongInt);
         2: (ints: ARRAY[0..1] OF Integer);
   END;
```

Listing 2-6 illustrates how to use this technique to determine the amount of free space on a volume (specified by its volume reference number).

Listing 2-6 Determining the amount of free space on a volume

```
FUNCTION GetVolumeFreeSpace (myVol: Integer): LongInt;
VAR
   myHPB:   HParamBlockRec;      {parameter block for PBHGetVInfo}
   myErr:   OSErr;               {result code from PBHGetVInfo}
   myRec:   TwoIntsMakeALong;    {easy way to get an unsigned int}
BEGIN
   WITH myHPB DO
      BEGIN
         ioNamePtr := NIL;
         ioVRefNum := myVol;
         ioVolIndex := 0;
      END;
   myErr := PBHGetVInfo(@myHPB, FALSE);
   IF myErr = noErr THEN
      BEGIN
         myRec.ints[0] := 0;
         myRec.ints[1] := myHPB.ioVFrBlk;
         GetVolumeFreeSpace := myRec.long * myHPB.ioVAlBlkSiz;
      END
   ELSE
      GetVolumeFreeSpace := 0;
END;
```

If the value passed to GetVolumeFreeSpace is a valid volume reference number, then this function reads the number of free allocation blocks on the volume, installs that number as the low-order word of a long integer, and performs the necessary multiplication to determine how many bytes are free on the volume.

Note

You could avoid these complications with unsigned integers by calling `PBHGetVInfo` as illustrated and then passing the value returned in the `ioVDrvInfo` field to the high-level function `GetVInfo`. The technique using the `TwoIntsMakeALong` data type to convert unsigned integers to long integers is illustrated here because it is useful when reading the fields of many other File Manager data structures from Pascal. For example, the `vcbFreeBks` field of a volume control block contains an unsigned integer that you can interpret in this way. ◆

Sharing Volumes and Directories

The File Manager includes several functions that allow you to manipulate share points on local volumes that have file sharing enabled and to obtain a list of user and group names and IDs recognized by the local file server. These functions are especially useful if you need to implement a dialog box that allows the user to designate a volume or directory as a share point or to set the owner, user, and group of a shared folder.

The `PBShare` function makes a volume or directory a share point, hence available on the network. The `PBUnshare` function undoes the effects of `PBShare`: it makes an existing share point unavailable on the network. The `PBGetUGEntry` function lets you create a list of user and group names and IDs on the local server.

Before calling any of these functions, you should check whether file sharing is enabled on the local machine and, if so, whether the desired local volume is sharable. You can determine whether a particular volume is sharable by using the function `VolIsSharable` defined in Listing 2-7.

Listing 2-7 Determining whether a volume is sharable

```
FUNCTION VolIsSharable (vRefNum: Integer): Boolean;
VAR
   myHPB:         HParamBlockRec;
   myInfoBuffer:  GetVolParmsInfoBuffer;
   myErr:         OSErr;
BEGIN
WITH myHPB DO
   BEGIN
      ioNamePtr := NIL;
      ioVRefNum := vRefNum;
      ioBuffer := @myInfoBuffer;
      ioReqCount := SizeOf(myInfoBuffer);
   END;
   myErr := PBHGetVolParms(@myHPB, FALSE);
   IF myErr = noErr THEN
      IF BTst(myInfoBuffer.vMAttrib, bHasPersonalAccessPrivileges) THEN
```

```
         VolIsSharable := TRUE
      ELSE
         VolIsSharable := FALSE
   ELSE
      VolIsSharable := FALSE;
END;
```

The `VolIsSharable` function inspects the `bHasPersonalAccessPrivileges` bit returned in the `vMAttrib` field of the volume attributes buffer it passed to `PBHGetVolParms`. If this bit is set, local file sharing is enabled on the specified volume.

You can use the function `SharingIsOn` defined in Listing 2-8 to determine whether file sharing is enabled on the local machine.

Listing 2-8 Determining whether file sharing is enabled

```
FUNCTION SharingIsOn: Boolean;
VAR
   myHPB:      HParamBlockRec;
   myErr:      OSErr;
   volIndex:   Integer;
   sharing:    Boolean;
BEGIN
   sharing := FALSE;                {assume file sharing is off}
   volIndex := 1;
   REPEAT
      WITH myHPB DO
         BEGIN
            ioNamePtr := NIL;
            ioVolIndex := volIndex;
         END;
      myErr := PBHGetVInfo(@myHPB, FALSE);
      IF myErr = noErr THEN
         sharing := VolIsSharable(myHPB.ioVRefNum);
      volIndex := volIndex + 1;
   UNTIL (myErr <> noErr) OR sharing;
   SharingIsOn := sharing;
END;
```

The `SharingIsOn` function simply calls the `VolIsSharable` function for each local volume (or until a sharable volume is found). It uses indexed calls to `PBHGetVInfo` to obtain the volume reference number of each mounted volume.

Locking and Unlocking File Ranges

A file can be opened with shared read / write permission to allow several users to share the data in the file. When a user needs to modify a portion of a file that has been opened with shared read / write permission, it is usually desirable to make that portion of the file unavailable to other users while the changes are made. You can call the `PBLockRange` function to lock a range of bytes before modifying the file and then `PBUnlockRange` to unlock that range after your changes are safely recorded in the file.

Locking a range of bytes in a file gives the user exclusive read / write access to that range and makes it inaccessible to other users. Other users can neither write nor read the bytes in that range until you unlock it. If other users attempt to read data from a portion of a file that you have locked, they receive the `fLckdErr` result code.

The functions `PBLockRange` and `PBUnlockRange` are effective only on files that are located on volumes that are sharable. If you call `PBLockRange` on a file that is not located on a remote server volume or that is not currently being shared, no range locking occurs. Moreover, `PBLockRange` does not return a result code indicating that no range locking has occurred. As a result, you should usually check whether range locking will be effective on a file before attempting to lock the desired range.

Listing 2-9 illustrates how you can check to make sure that calling `PBLockRange` will have the desired effect.

Listing 2-9 Determining whether a file can have ranges locked

```
FUNCTION RangesCanBeLocked (fRefNum: Integer): Boolean;
VAR
   myParmBlk:  ParamBlockRec;             {basic parameter block}
   myErr:      OSErr;
BEGIN
   WITH myParmBlk DO
      BEGIN
         ioRefNum := fRefNum;
         ioReqCount := 1;                 {lock a single byte}
         ioPosMode := fsFromStart;        {at the beginning of the file}
         ioPosOffset := 0;
      END;
   myErr := PBLockRange(@myParmBlk, FALSE);{lock the byte; ignore result}
   myErr := PBLockRange(@myParmBlk, FALSE);{lock the byte again}


   CASE myErr OF
      fLckdErr,                           {byte was locked by another user}
      afpRangeOverlap,                    {byte was locked by this user}
      afpNoMoreLocks:                     {max number of locks already used}
```

```
         BEGIN
            RangesCanBeLocked := TRUE;    {range locking is supported}
            IF myErr = afpRangeOverlap THEN  {unlock the byte we locked}
               myErr := PBUnlockRange(@myParmBlk, FALSE);
         END;
      OTHERWISE
         RangesCanBeLocked := FALSE;      {range locking is not supported}
   END; {of CASE}
END;
```

The function `RangesCanBeLocked` takes a file reference number of an open file as a parameter; this is the reference number of the file in which a range of bytes is to be locked. The function attempts to locks the first byte in the file and immediately attempts to lock it again. If the second range locking fails with the result code `afpRangeOverlap`, the first call to `PBLockRange` was successful. If the second call to `PBLockRange` fails with the result code `fLckdErr`, the byte was already locked by another user. Similarly, if the second call to `PBLockRange` fails with the result code `afpNoMoreLocks`, the maximum number of range locks has been reached. In these three cases, range locking is supported by the volume containing the specified file. If any other result code (including `noErr`) is returned, range locking is not supported by that volume or for some reason the capabilities of the volume cannot be determined.

Note
Local file sharing can be started or stopped (via the Sharing Setup control panel) while your application is running. For this reason, each time you want to lock a range, it's best to check that byte ranges in that file can be locked. ◆

You can unlock a locked range of bytes by calling `PBUnlockRange`. Note that the range to be unlocked must be the exact same range of bytes that was previously locked using `PBLockRange`. (You can lock and unlock different byte ranges in any order, however.) If for some reason you need to unlock a range of bytes and do not know where the range started or how long the range is, you must close the file to unlock the range. When a file is closed, all locked ranges held by a user are unlocked.

If you want to append data to a shared file, you can use `PBLockRange` to lock the range of bytes from the file's current logical end-of-file to the last possible addressable byte of the file. Once you have locked that range, you can write data into it. Listing 2-10 shows how to determine the current logical end-of-file and lock the appropriate range.

Listing 2-10 Locking a file range to append data to the file

```
FUNCTION LockRangeForAppending (fRefNum: Integer; VAR EOF: LongInt): OSErr;
VAR
   myParmBlk:  ParamBlockRec;             {basic parameter block}
   myErr:      OSErr;
   myEOF:      LongInt;                   {current EOF}
```

```
BEGIN
   myParmBlk.ioCompletion := NIL;
   myParmBlk.ioRefNum := fRefNum;
   myErr := PBGetEOF(@myParmBlk, FALSE);  {get the current EOF}
   IF myErr <> noErr THEN
      BEGIN
         LockRangeForAppending := myErr;
         Exit(LockRangeForAppending);     {trouble reading EOF}
      END;
   myEOF := LongInt(myParmBlk.ioMisc);    {save the current EOF}
   WITH myParmBlk DO
      BEGIN
         ioReqCount := -1;                {all addressable bytes}
         ioPosMode := fsFromStart;        {start range...}
         ioPosOffset := myEOF;            {...at the current end-of-file}
      END;
   myErr := PBLockRange(@myParmBlk, FALSE);{lock the specified range}
   EOF := myEOF;                          {return current EOF to caller}
   LockRangeForAppending := myErr;
END;
```

The function `LockRangeForAppending` first determines the current logical end-of-file. It is important to get this value immediately before you attempt to lock a range that depends on it because another user of the shared file might have changed the end-of-file since you last read it. Then `LockRangeForAppending` locks the range beginning at the current end-of-file and extending for the maximum number of bytes (specified using the special value –1).

In effect, this technique locks a range where data does not yet exist. Practically speaking, locking the entire addressable range of a file prevents another user from appending data to the file until you unlock that range. Note that `LockRangeForAppending` returns the current logical end-of-file to the caller so that the caller can unlock the correct range of bytes after appending the data.

You can also call `PBLockRange` to lock a range of bytes when you want to truncate a file. Locking the end portion of a file to be deleted prevents another user from using that portion during the truncation. Instead of setting the `ioPosOffset` field of the parameter block to the logical end-of-file (as in Listing 2-10), simply set it to what will be the last byte after the file is truncated. Similarly, you can lock an entire file fork by setting the `ioPosOffset` field to 0.

Data Organization on Volumes

This section describes how data is organized on HFS volumes. In general, an application that simply manipulates data stored in files does not need to know how that data is organized on a volume or on the physical storage medium containing that volume. The

organization described in this section is maintained by the File Manager for its own uses. Some specialized applications and file-system utilities, however, do need to know exactly how file data is stored on a disk.

▲ **WARNING**

This section is provided primarily for informational purposes. The organization of data on volumes is subject to change. Before you use this information to read or modify the data stored on a volume, be sure to check that the `drSigWord` field in the master directory block (described in "Master Directory Blocks" beginning on page 2-60) identifies that volume as an HFS volume. ▲

Much of the information describing the files and directories on an HFS volume is read into memory when the volume is mounted. (For example, most of the volume's master directory block is read into memory as a volume control block.) For a description of how that data is organized in memory, see "Data Organization in Memory" beginning on page 2-77.

The File Manager uses a number of interrelated structures to manage the organization of data on disk and in memory. For this reason, it is easy to lose sight of the simple and elegant scheme that underlies these structures. As you read through this section and the next, you should keep these points in mind:

- The File Manager keeps track of which blocks on a disk are allocated to files and which are not by storing a *volume bitmap* on disk and in memory. If a bit in the map is set, the corresponding block is allocated to some file; otherwise, the corresponding block is free for allocation.
- The File Manager always allocates logical disk blocks to a file in groups called *allocation blocks*; an allocation block is simply a group of consecutive logical blocks. The size of a volume's allocation blocks depends on the capacity of the volume; there can be at most 65,535 allocation blocks on a volume.
- The File Manager keeps track of the directory hierarchy on a volume by maintaining a file called the *catalog file*; the catalog file lists all the files and directories on a volume, as well as some of the attributes of those files and directories. A catalog file is organized as a B*-tree (or "balanced tree") to allow quick and efficient searches through a directory hierarchy that is typically quite large.
- The File Manager keeps track of which allocation blocks belong to a file by maintaining a list of the file's extents; an *extent* is a contiguous range of allocation blocks allocated to some file, which can be represented by a pair of numbers: the start of the range and the length of the range. The first three extents of most files are stored in the volume's catalog file. All remaining file extents are stored in the *extents overflow file*, which is also organized as a B*-tree.
- The first three extents of the catalog file and the extents overflow file are stored in the master directory block (on disk) and the volume control buffer (in memory); a master directory block is always located at a fixed offset from the beginning of a volume, and a volume control block is stored in the VCB queue.

Disk and Volume Organization

A **disk** is a physical medium capable of storing information. Examples of disks include 3.5-inch floppy disks, SCSI hard disks and CD-ROM discs, and even RAM disks. A SCSI disk may be divided into one or more partitions. A **partition** is simply part of a disk that has been allocated to a particular operating system, file system, or device driver. For example, you can partition a single SCSI disk into both Macintosh partitions and A/UX partitions. The Macintosh partitions are typically used to hold Macintosh volumes. An A/UX partition can contain an A/UX file system, but it can also be used as a paging area for virtual memory or as a storage area for autorecovery files.

The information describing the division of a SCSI disk into partitions is contained in the disk's **partition map,** which is always located in the first physical block (512 bytes) on a disk. The partition map specifies the first and last physical blocks in each partition, as well as additional information about the partition (such as its type). The exact structure of a partition map is described in the chapter "SCSI Manager" in *Inside Macintosh: Devices*.

Often the first partition on a SCSI disk, following the partition map, is the driver partition that contains the actual device driver used to communicate with the disk. (There is, however, no requirement that the driver partition be the first partition on a disk.) Figure 2-4 illustrates a typical organization of partitions on a disk.

A partition can contain at most one volume. A **volume** is a single disk partition that contains both file data and the file and directory information necessary to maintain the appropriate data organization or file system. For example, a volume can contain a Macintosh, ProDOS, MS-DOS, or A/UX file system structure. Notice in Figure 2-4 that a Macintosh volume occupies only part of the entire physical disk, and that there can be multiple partitions (both Macintosh volumes or other types of partitions) on a given disk.

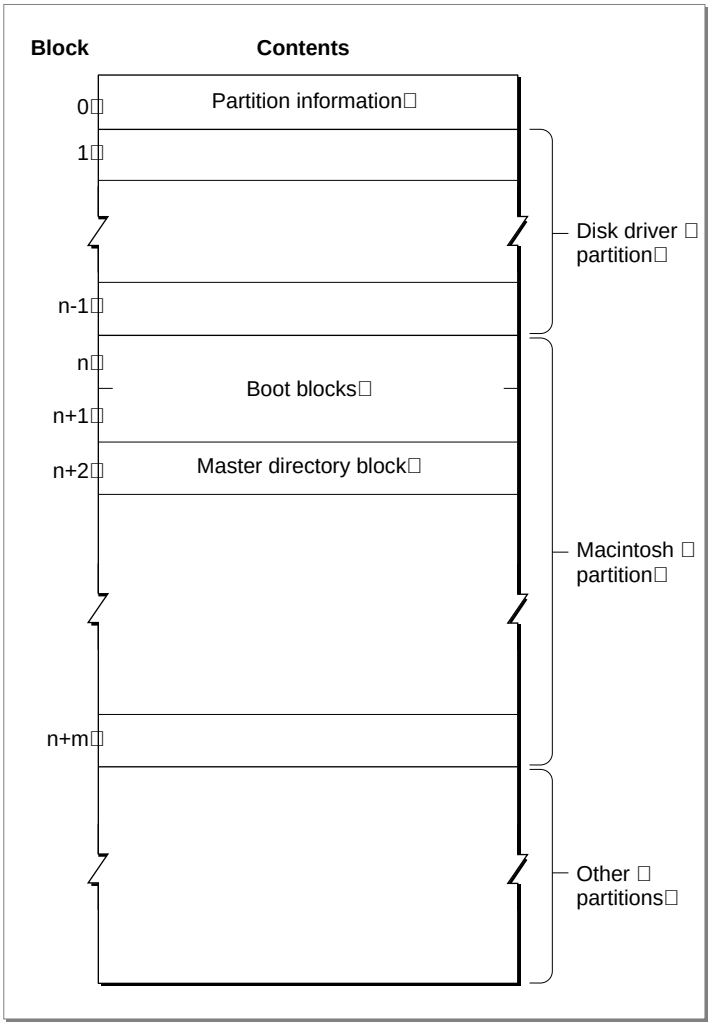
Note
The disk organization illustrated in Figure 2-4 does not apply to Macintosh 3.5-inch floppy disks. Because each floppy disk is one volume, there is no need for a disk partition map. Also, there is no device driver partition on a floppy disk. ◆

The remainder of this section describes only **HFS volumes,** that is, Macintosh file systems organized using the hierarchical file system (HFS) implemented on the Macintosh Plus and later models.

Each HFS volume begins with two boot blocks. The boot blocks on the startup volume are read at system startup time and contain booting instructions and other important information such as the name of the System file and the Finder. Following the boot blocks are two additional structures, the master directory block and the volume bitmap.

The master directory block contains information about the volume, such as the date and time of the volume's creation and the number of files on the volume. The volume bitmap contains a record of which blocks in the volume are currently in use.

Figure 2-4 Organization of partitions on a disk

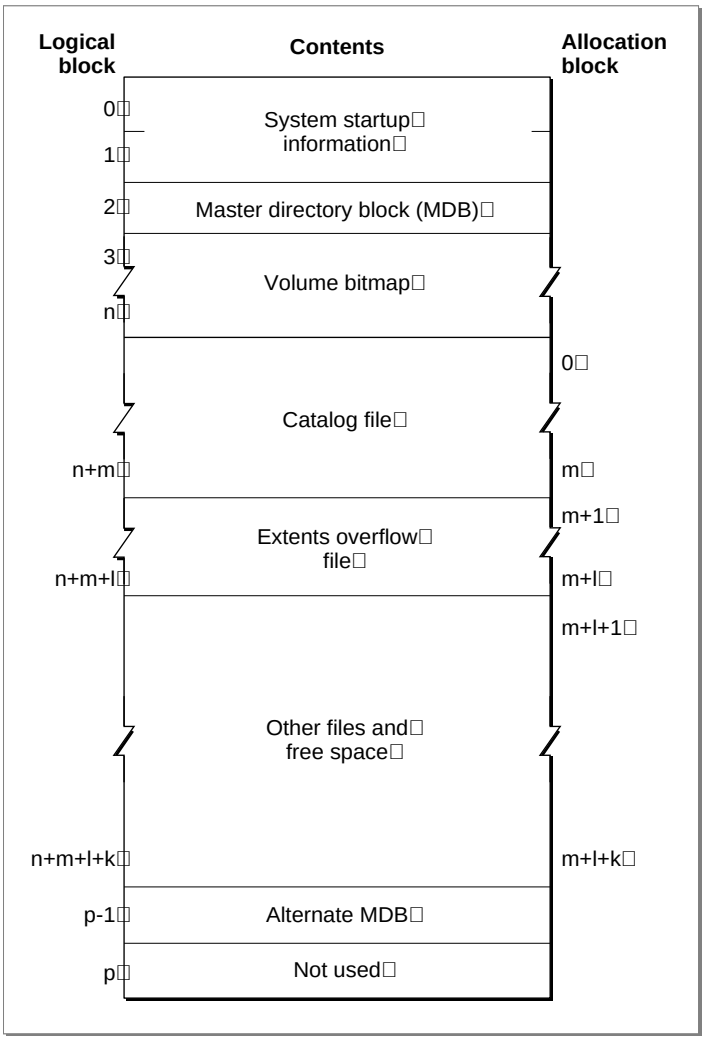


The largest portion of a volume consists of four types of information or areas:

- applications and data files
- the catalog file
- the extents overflow file
- unused space

The general structure of an HFS volume is illustrated in Figure 2-5.

Figure 2-5 Organization of a volume



All the areas on a volume are of fixed size and location, except for the catalog file and the extents overflow file. These two files can appear anywhere between the volume bitmap and the alternate master directory block (MDB). They can appear in any order and are not necessarily contiguous.

The information on all block-formatted volumes is organized in logical blocks and allocation blocks. Logical blocks contain a number of bytes of standard information (512 bytes on Macintosh-initialized volumes). Allocation blocks are composed of any integral number of logical blocks and are simply a means of grouping logical blocks in more convenient parcels. The allocation block size is a volume parameter whose value is set when the volume is initialized; it cannot be changed unless the volume is reinitialized.

To promote file contiguity and avoid fragmentation, space is allocated to files in groups of allocation blocks, or **clumps.** The clump size is always a multiple of the allocation

block size, and it's the minimum number of bytes to allocate each time the `Allocate` function is called or the physical end-of-file is reached during a write operation. The clump size is specified in the catalog information for a file; you can determine the clump size using the `PBGetCatInfo` function.

The rest of this section describes in detail the structure of the boot blocks, the master directory block, and the catalog and extents overflow files. It also describes the general structure of a B*-tree, because the catalog and extents overflow files are both organized as B*-trees.

Boot Blocks

The first two logical blocks on every Macintosh volume are **boot blocks.** These blocks contain **system startup information:** instructions and information necessary to start up (or "boot") a Macintosh computer. This information consists of certain configurable system parameters (such as the capacity of the event queue, the number of open files allowed, and so forth) and is contained in a boot block header. The system startup information also includes actual machine-language instructions that could be used to load and execute the System file. Usually these instructions follow immediately after the boot block header. Generally, however, the boot code stored on disk is ignored in favor of boot code stored in a resource in the System file.

The structure of the boot block header can be described by the Pascal `BootBlkHdr` data type.

▲ **WARNING**
The format of the boot block header is subject to change. If your application relies on the information presented here, it should check the boot block header version number and react gracefully if that number is greater than that documented here. ▲

Note that there are two boot block header formats. The current format includes two fields at the end that are not contained in the older format. These fields allow the Operating System to size the System heap relative to the amount of available physical RAM. A boot block header that conforms to the older format sets the size of the System heap absolutely, using values specified in the header itself. You can determine whether a boot block header uses the current or the older format by inspecting a bit in the high-order byte of the `bbVersion` field, as explained in its field description.

```
TYPE BootBlkHdr   =           {boot block header}
RECORD
   bbID:             Integer; {boot blocks signature}
   bbEntry:          LongInt; {entry point to boot code}
   bbVersion:        Integer; {boot blocks version number}
   bbPageFlags:      Integer; {used internally}
   bbSysName:        Str15;   {System filename}
   bbShellName:      Str15;   {Finder filename}
   bbDbg1Name:       Str15;   {debugger filename}
```

```
   bbDbg2Name:       Str15;   {debugger filename}
   bbScreenName:     Str15;   {name of startup screen}
   bbHelloName:      Str15;   {name of startup program}
   bbScrapName:      Str15;   {name of system scrap file}
   bbCntFCBs:        Integer; {number of FCBs to allocate}
   bbCntEvts:        Integer; {number of event queue elements}
   bb128KSHeap:      LongInt; {system heap size on 128K Mac}
   bb256KSHeap:      LongInt; {used internally}
   bbSysHeapSize:    LongInt; {system heap size on all machines}
   filler:           Integer; {reserved}
   bbSysHeapExtra:   LongInt; {additional system heap space}
   bbSysHeapFract:   LongInt; {fraction of RAM for system heap}
END;
```

Field descriptions

`bbID` A signature word. For HFS volumes, this field always contains the value $4C4B.

`bbEntry` The entry point to the boot code stored in the boot blocks. This field contains machine-language instructions that translate to `BRA.S *+$90` (or `BRA.S *+$88`, if the older block header format is used), which jumps to the main boot code following the boot block header. This field is ignored, however, if bit 6 is clear in the high-order byte of the `bbVersion` field or if the low-order byte in that field contains $D.

`bbVersion` A flag byte and boot block version number. The high-order byte of this field is a flag byte whose bits have the following meanings:

Bit	Meaning
0–4	Reserved; must be 0
5	Set if relative system heap sizing is to be used
6	Set if the boot code in boot blocks is to be executed
7	Set if new boot block header format is used

If bit 7 is clear, then bits 5 and 6 are ignored and the version number is found in the low-order byte of this field. If that byte contains a value that is less than $15, the Operating System ignores any values in the `bb128KSHeap` and `bb256KSHeap` fields and configures the System heap to the default value contained in the `bbSysHeapSize` field. If that byte contains a value that is greater than or equal to $15, the Operating System sets the System heap to the value in `bbSysHeapSize`. In addition, the Operating System executes the boot code in the `bbEntry` field only if the low-order byte contains $D.

If bit 7 is set, the Operating System inspects bit 6 to determine whether to execute the boot code contained in the `bbEntry` field and bit 5 to determine whether to use relative System heap sizing. If bit 5 is clear, the Operating System sets the System heap to the value

	in `bbSysHeapSize`. If bit 5 is set, the System heap is extended by the value in `bbSysHeapExtra` plus the fraction of available RAM specified in `bbSysHeapFract`.
`bbPageFlags`	Used internally.
`bbSysName`	The name of the System file.
`bbShellName`	The name of the shell file. Usually, the system shell is the Finder.
`bbDbg1Name`	The name of the first debugger installed during the boot process. Typically this is Macsbug.
`bbDbg2Name`	The name of the second debugger installed during the boot process. Typically this is Disassembler.
`bbScreenName`	The name of the file containing the startup screen. Usually this is StartUpScreen.
`bbHelloName`	The name of the startup program. Usually this is Finder.
`bbScrapName`	The name of the system scrap file. Usually this is Clipboard.
`bbCntFCBs`	The number of file control blocks (FCBs) to put in the FCB buffer. In system software version 7.0 and later, this field specifies only the initial number of FCBs in the FCB buffer, because the Operating System can usually resize the FCB buffer if necessary. See “File Control Blocks” on page 2-82 for details on the FCB buffer.
`bbCntEvts`	The number of event queue elements to allocate. This number determines the maximum number of events that the Event Manager can store at any one time. Usually this field contains the value 20.
`bb128KSHeap`	The size of the System heap on a Macintosh computer having 128 KB of RAM.
`bb256KSHeap`	Reserved.
`bbSysHeapSize`	The size of the System heap on a Macintosh computer having 512 KB or more of RAM. This field might be ignored, as explained in the description of the `bbVersion` field.
`filler`	Reserved.
`bbSysHeapExtra`	The minimum amount of additional System heap space required. If bit 5 of the high-order word of the `bbVersion` field is set, this value is added to `bbSysHeapSize`.
`bbSysHeapFract`	The fraction of RAM available to be used for the System heap. If bit 5 of the high-order word of the `bbVersion` field is set, this fraction of available RAM is added to `bbSysHeapSize`.

Master Directory Blocks

A **master directory block** (MDB)—also sometimes known as a **volume information block** (VIB)—contains information about the rest of the volume. This information is written into the MDB when the volume is initialized. Thereafter, whenever the volume is mounted, the File Manager reads the information in the MDB and copies some of that information into a volume control block (VCB). A VCB is a private data structure maintained in memory by the File Manager (in the VCB queue). The structure of a VCB is described in “Volume Control Blocks,” later in this chapter.

Note in Figure 2-5 (page 2-57) that a copy of the MDB is located in the next-to-last block in the volume. This copy is updated only when the extents overflow file or the catalog file grows larger. This alternate MBD is intended for use solely by disk utilities.

The MDB data type defines a master directory block record.

```
TYPE MDB          =           {master directory block}
RECORD
   drSigWord:     Integer;    {volume signature}
   drCrDate:      LongInt;    {date and time of volume creation}
   drLsMod:       LongInt;    {date and time of last modification}
   drAtrb:        Integer;    {volume attributes}
   drNmFls:       Integer;    {number of files in root directory}
   drVBMSt:       Integer;    {first block of volume bitmap}
   drAllocPtr:    Integer;    {start of next allocation search}
   drNmAlBlks:    Integer;    {number of allocation blocks in volume}
   drAlBlkSiz:    LongInt;    {size (in bytes) of allocation blocks}
   drClpSiz:      LongInt;    {default clump size}
   drAlBlSt:      Integer;    {first allocation block in volume}
   drNxtCNID:     LongInt;    {next unused catalog node ID}
   drFreeBks:     Integer;    {number of unused allocation blocks}
   drVN:          String[27]; {volume name}
   drVolBkUp:     LongInt;    {date and time of last backup}
   drVSeqNum:     Integer;    {volume backup sequence number}
   drWrCnt:       LongInt;    {volume write count}
   drXTClpSiz:    LongInt;    {clump size for extents overflow file}
   drCTClpSiz:    LongInt;    {clump size for catalog file}
   drNmRtDirs:    Integer;    {number of directories in root directory}
   drFilCnt:      LongInt;    {number of files in volume}
   drDirCnt:      LongInt;    {number of directories in volume}
   drFndrInfo:    ARRAY[1..8] OF LongInt;
                              {information used by the Finder}
   drVCSize:      Integer;    {size (in blocks) of volume cache}
   drVBMCSize:    Integer;    {size (in blocks) of volume bitmap cache}
   drCtlCSize:    Integer;    {size (in blocks) of common volume cache}
   drXTFlSize:    LongInt;    {size of extents overflow file}
   drXTExtRec:    ExtDataRec; {extent record for extents overflow file}
   drCTFlSize:    LongInt;    {size of catalog file}
   drCTExtRec:    ExtDataRec; {extent record for catalog file}
END;
```

Field descriptions

`drSigWord` The volume signature. For HFS volumes, this field contains $4244; for the obsolete flat MFS volumes, this field contains $D2D7.

`drCrDate` The date and time of volume creation (initialization).

`drLsMod` The date and time the volume was last modified. This is not necessarily when the volume was last flushed.

`drAtrb` Volume attributes. Currently the following bits are defined:

Bit	Meaning
7	Set if the volume is locked by hardware
8	Set if the volume was successfully unmounted
9	Set if the volume has had its bad blocks spared
15	Set if the volume is locked by software

`drNmFls` The number of files in the root directory.

`drVBMSt` The first block of the volume bitmap. This field always contains 3 in the current implementation.

`drAllocPtr` The number of the allocation block at which the next allocation search will begin. Used internally.

`drNmAlBlks` The number of allocation blocks in the volume. Because the value in this field is an integer, a volume can contain at most 65,535 allocation blocks.

`drAlBlkSiz` The allocation block size (in bytes). This value must always be a multiple of 512 bytes.

`drClpSiz` The default clump size.

`drAlBlSt` The location of the first allocation block in the volume.

`drNxtCNID` The next unused catalog node ID (directory ID or file ID).

`drFreeBks` The number of unused allocation blocks on the volume.

`drVN` The volume name. This field consists of a length byte followed by 27 bytes. Note that the volume name can occupy at most 27 characters; this is an exception to the normal file and directory name limit of 31 characters.

`drVolBkUp` The date and time of the last volume backup.

`drVSeqNum` Volume backup sequence number. Used internally.

`drWrCnt` The volume write count (that is, the number of times the volume has been written to).

`drXTClpSize` The clump size for the extents overflow file.

`drCTClpSize` The clump size for the catalog file.

`drNmRtDirs` The number of directories in the root directory.

`drFilCnt` The number of files on the volume.

`drDirCnt` The number of directories on the volume.

`drFndrInfo` Information used by the Finder. See the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials* for details on Finder information.

`drVCSize` The size (in allocation blocks) of the volume cache. Used internally.

`drVBMCSize` The size (in allocation blocks) of the volume bitmap cache. Used internally.

`drCtlCSize` The size (in allocation blocks) of the common volume cache. Used internally.

`drXTFlSize`	The size (in allocation blocks) of the extents overflow file.
`drXTExtRec`	First extent record for the extents overflow file. An extent record is an array of three extents. See “Extents Overflow Files” on page 2-75 for a description of extents and extent records.
`drCTFlSize`	The size (in allocation blocks) of the catalog file.
`drCTExtRec`	First extent record for the catalog file.

Note
The values in the `drNmAlBlks` and `drFreeBks` fields should be interpreted as unsigned integers (that is, they can range from 0 to 65,535, not from –32,768 to 32,767). Pascal does not support unsigned data types, and so you need to use the technique illustrated in “Determining the Amount of Free Space on a Volume” on page 2-47 to read the values in these fields correctly. ◆

Volume Bitmaps

The File Manager uses a **volume bitmap** to keep track of whether each block in a volume is currently allocated to some file or not. The bitmap contains one bit for each allocation block in the volume. If a bit is set, the corresponding allocation block is currently in use by some file. If a bit is clear, the corresponding allocation block is not currently in use by any file and is available for allocation.

Note
The volume bitmap indicates which blocks on a volume are currently in use, but it does not indicate which files occupy which blocks. The File Manager maintains file-mapping information in two locations: in each file’s catalog entry and in the extents overflow file. ◆

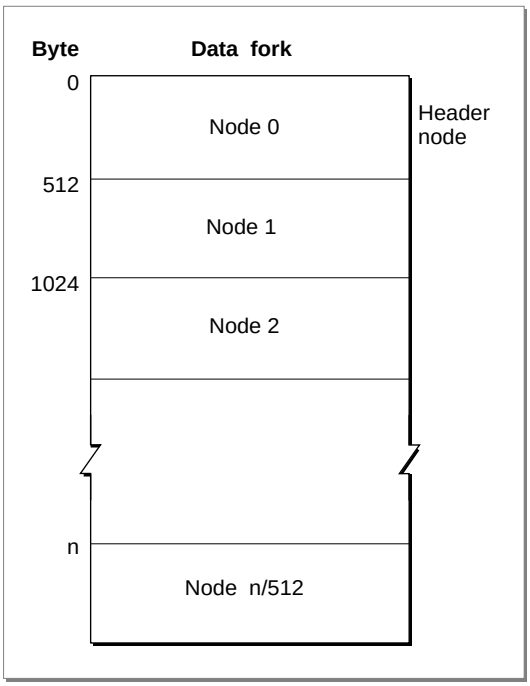
The size of the volume bitmap depends on the number of allocation blocks in the volume, which in turn depends both on the number of physical blocks in the volume and on the size of the volume’s allocation blocks (the number of physical blocks per allocation block). For example, a floppy disk that can hold 800 KB of data and has an allocation block size of one physical block has a volume bitmap size of 1600 bits (200 bytes). A volume containing 32 MB of data and having an allocation block size of one physical block has a volume bitmap size of 65,536 bits (8192 bytes). However, the size of the volume bitmap is rounded up, if necessary, so that the volume bitmap occupies an integral number of physical blocks.

Because the `drNmAlBlks` field in the MDB occupies only 2 bytes, the File Manager can address at most 65,535 allocation blocks. Thus, the volume bitmap is never larger than 8192 bytes (or 16 physical blocks). For volumes containing more than 32 MB of space, the allocation block size must be increased. For example, a volume containing 40 MB of space must have an allocation block size that is at least 2 physical blocks; a volume containing 80 MB of space must have an allocation block size that is at least 3 physical blocks; and so forth.

B*-Trees

The File Manager maintains information about a volume's directory hierarchy and file block mapping in two files that are organized as B*-trees to allow quick and efficient retrieval of that information. In a **B*-tree,** all the information that needs to be stored is intelligently classified and sorted into objects called nodes. Figure 2-6 illustrates the general structure of a B*-tree file.

Figure 2-6 The structure of a B*-tree file



Note that each B*-tree file used by the File Manager makes use of the data fork only; the resource fork of a B*-tree file is unused. The length of a B*-tree file varies according to the number of nodes it contains.

A node in turn contains records, which can be used for a variety of purposes. Some records contain the actual data that is to be retrieved and possibly updated; these records occupy nodes called leaf nodes. Other records contain information about the structure of the B*-tree. The File Manager uses these records to find the information it needs quickly. There are three types of these "bookkeeping" nodes: header nodes, index nodes, and map nodes.

Nodes

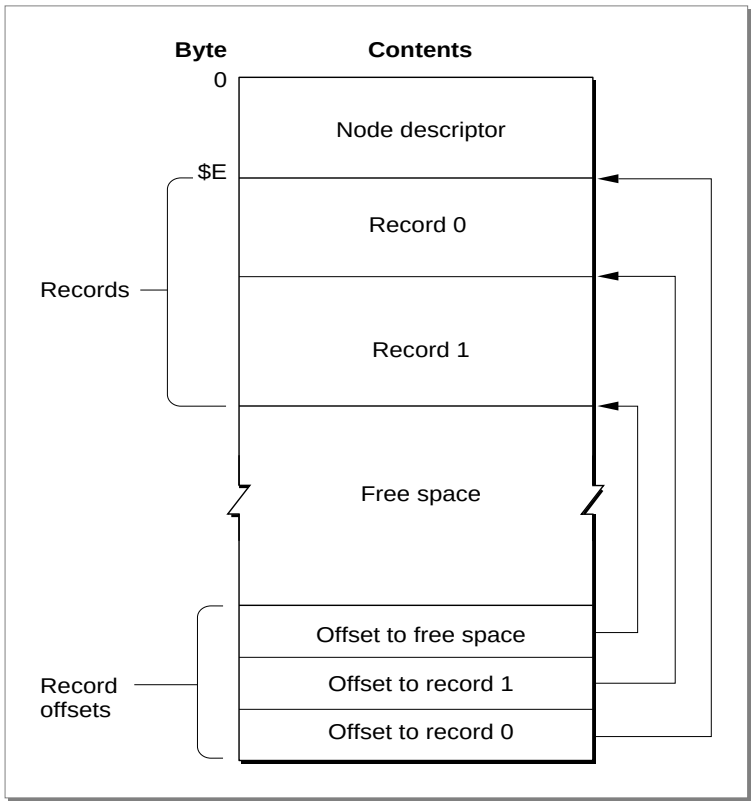
A B*-tree file consists entirely of objects called **nodes,** each of which is 512 bytes long. Figure 2-7 illustrates the structure of a node.

Each node has the same general structure and consists of three main parts: a node descriptor that starts at the beginning of the node, a group of record offsets that starts at the end of the node, and a group of records.

The **node descriptor** contains information about the node, as well as forward and backward links to other nodes. You can use the `NodeDescriptor` data type to display the structure of a node descriptor.

```
TYPE NodeDescriptor  =          {node descriptor}
RECORD
   ndFLink:       LongInt;      {forward link}
   ndBLink:       LongInt;      {backward link}
   ndType:        SignedByte;   {node type}
   ndNHeight:     SignedByte;   {node level}
   ndNRecs:       Integer;      {number of records in node}
   ndResv2:       Integer;      {reserved}
END;
```

Figure 2-7 The structure of a node



Field descriptions

`ndFLink`	A link to the next node of this type. If this node is the last node, this field contains `NIL`.
`ndBLink`	A link to the previous node of this type. If this node is the first node, this field contains `NIL`.
`ndType`	The type of this node. Currently four types of nodes are recognized, defined by the constants listed in this section.
`ndNHeight`	The level or "depth" of this node in the B*-tree hierarchy. The top-level node (a header node, described in "Header Nodes" on page 2-68) always has a level of 0; all other nodes have a level that is one greater than their parent node. Currently, the maximum depth of a node is 8.
`ndNRecs`	The number of records contained in this node.
`ndResv2`	Reserved. This field should always be 0.

A node descriptor is always $0E bytes in length, and so the records contained in the node always begin at offset $0E from the beginning of the node. The size of a record can vary, depending on its type and on the amount of information it contains; as a result, the File Manager accesses a record by storing the offset from the beginning of the node to that record in the list of offsets found at the end of the node. Each offset occupies a word, and (as you might have guessed) the last word in a node always contains the value $0E, pointing to the first record in the node. The offsets to subsequent records are stored in order starting from the end of the node, as illustrated in Figure 2-7.

Note that there is always one more offset than the number of records contained in a node; this is an offset to the beginning of any unused space in the node. If there is no free space in the node, then that offset contains its own byte offset within the node.

The `ndType` field of the node descriptor indicates the type of a node. In essence, the type of a node indicates what kinds of records it contains and hence what its function in the B*-tree hierarchy is. The File Manager maintains four kinds of nodes in a B*-tree, indicated by constants:

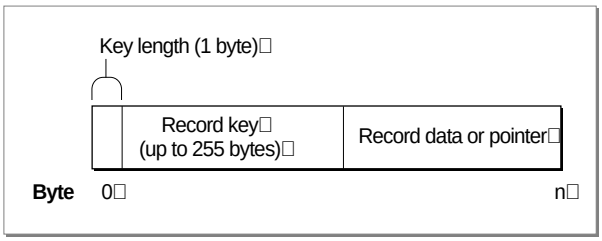
```
CONST                           {node types}
   ndIndxNode     =  $00;       {index node}
   ndHdrNode      =  $01;       {header node}
   ndMapNode      =  $02;       {map node}
   ndLeafNode     =  $FF;       {leaf node}
```

These node types are described in the four sections immediately after the next one.

Node Records

A record in a B*-tree node contains either data or a pointer to some other node in the tree. Figure 2-8 shows the general structure of a record in a leaf or index node.

Figure 2-8 Structure of a B*-tree node record



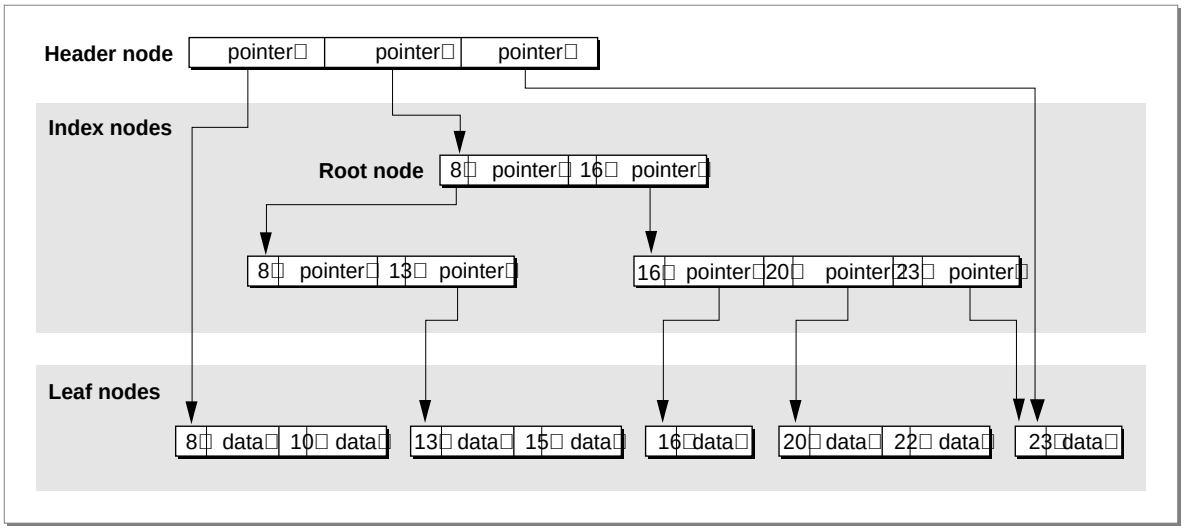
Note
The three records in a B*-tree header node do not have the structure depicted in Figure 2-8. They consist solely of data, as described in the next section, "Header Nodes." Similarly, the single record in a map node consists solely of data; see "Map Nodes" on page 2-70 for details. ◆

Each record contains a **search key,** which the File Manager uses to search through the B*-tree to locate the information it needs. The key can contain any information at all that is deemed useful in finding the data contained in the leaf nodes. In a catalog file, which maintains information about the hierarchy of files and directories on a volume, the search key is a combination of the file or directory name and the parent directory ID of that file or directory. In an extents overflow file, which maintains information about the extra extents belonging to a file, the search key is a combination of that file's type, its file ID, and the index of the first allocation block in the extent.

In a B*-tree, the records in each node are always grouped so that their keys are in ascending order. Moreover, the nodes on any given level are linked (through the `ndFLink` and `ndBLink` fields of their node descriptors) in such a way as to preserve the ascending order of record keys throughout that level. This is the essential ordering principle that allows the File Manager to search quickly through a tree. To illustrate this ordering scheme, Figure 2-9 shows a sample B*-tree containing hypothetical search keys (in this case, the keys are simply integers).

When the File Manager needs to find a data record, it begins searching at the root node (which is an index node, unless the tree has only one level), moving from one record to the next until it finds the record with the highest key that is less than or equal to the search key. The pointer of that record leads to another node, one level down in the tree. This process continues until the File Manager reaches a leaf node; then the records of that leaf node are examined until the desired key is found. At that point, the desired data has also been found.

Figure 2-9 A sample B*-tree



There is of course no guarantee that a record having the desired key will always be found in a search through a B*-tree. In this case, the search stops when a key larger than the search key is reached. (This is most likely to happen in a search through the catalog file.)

Header Nodes

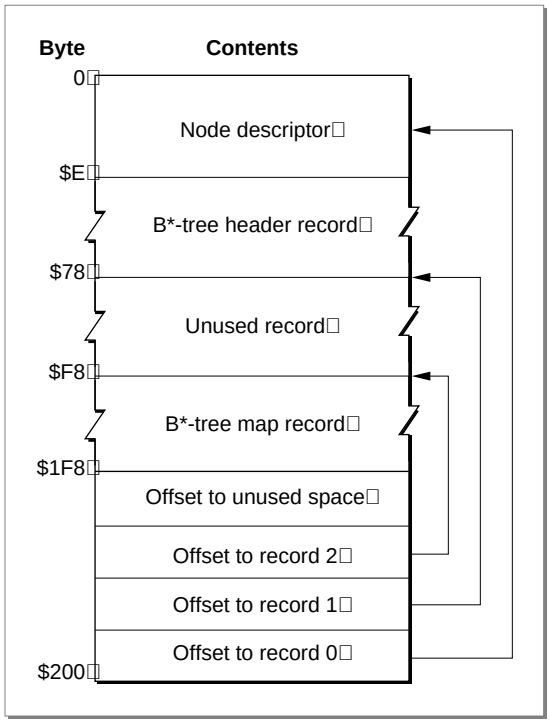
The first node (that is, node 0) in every B*-tree file is a **header node,** which contains essential information about the entire B*-tree file. The File Manager stores the location of the header node of the catalog file in the first 2 bytes of the `drCTExtRec` field of the MDB; the value in those 2 bytes indicates the allocation block number on which the catalog file (and hence the header node) begins. Similarly, the File Manager stores the location of the header node of the extents overflow file in the first 2 bytes of the `drXTExtRec` field of the MDB.

Note

When a volume is mounted, the File Manager reads the header node and copies some of the information it contains into a B*-tree control block in memory. See “B*-Tree Control Blocks” on page 2-84 for a description of this control block. ◆

A header node contains three records, the second of which occupies 128 bytes and is reserved for use by the File Manager. The other two records are called the B*-tree header record and the B*-tree map record; they occupy the first and third record positions, respectively. Hence, a header node has the structure illustrated in Figure 2-10.

Figure 2-10 Header node structure



Note

The three records contained in the header node do not contain keys. ◆

The **map record** is a bitmap that indicates which nodes in the B*-tree file are used and which are not. The bits are interpreted in exactly the same way as the bits in the volume bitmap: if a bit in the map record is set, then the corresponding node in the B*-tree file is being used. This bitmap occupies 256 bytes and can therefore encode information about 2048 nodes at most. If more nodes are needed to contain all the data that is to be stored in the B*-tree, the File Manager uses a map node to store additional mapping information. See the next section, “Map Nodes,” for a description of the structure of a map node.

The **B*-tree header record,** a data structure of type `BTHdrRec`, contains information about the beginning of the tree, as well as the size of the tree.

```
TYPE BTHdrRec     =          {B*-tree header}
RECORD
   bthDepth:      Integer;   {current depth of tree}
   bthRoot:       LongInt;   {number of root node}
   bthNRecs:      LongInt;   {number of leaf records in tree}
   bthFNode:      LongInt;   {number of first leaf node}
   bthLNode:      LongInt;   {number of last leaf node}
   bthNodeSize:   Integer;   {size of a node}
   bthKeyLen:     Integer;   {maximum length of a key}
```

```
   bthNNodes:     LongInt;    {total number of nodes in tree}
   bthFree:       LongInt;    {number of free nodes}
   bthResv:       ARRAY[1..76] OF SignedByte;   {reserved}
END;
```

Field descriptions

`bthDepth`	The current depth of the B*-tree.
`bthRoot`	The node number of the root node. The root node is the start of the B*-tree structure; usually the root node is first index node, but it might be a leaf node if there are no index nodes.
`bthNRecs`	The number of data records (records contained in leaf nodes).
`bthFNode`	The node number of the first leaf node.
`bthLNode`	The node number of the last leaf node.
`bthNodeSize`	The size (in bytes) of a node. Currently, this is always 512.
`bthKeyLen`	The maximum length of the key records in each node.
`bthNNodes`	The total number of nodes in the B*-tree.
`bthFree`	The total number of free nodes in the B*-tree.
`bthResv`	Reserved.

Map Nodes

As indicated in the previous section, the File Manager maintains a bitmap of the tree nodes in the map record of the B*-tree header node. If a B*-tree file contains more than 2048 nodes (enough for about 8000 files), the File Manager uses a **map node** to store additional node-mapping information. It stores the node number of the new map node in the `ndFLink` field of the node descriptor of the header node.

A map node consists of a node descriptor and a single map record. The map record is a continuation of the map record contained in the header node and occupies 494 bytes (512 bytes in the node, less 14 bytes for the node descriptor and 2 bytes for each of the two record offsets at the end of the node). A map node can therefore contain mapping information for an additional 3952 nodes.

If a B*-tree contains more than 6000 nodes (that is, 2048 + 3952, enough for about 25,000 files), the File Manager uses a second map node, the node number of which is stored in the `ndFLink` field of the node descriptor of the first map node. If more map nodes are required, each additional map node is similarly linked to the previous one.

Index Nodes

An **index node** contains records that point to other nodes in the B*-tree hierarchy. The File Manager uses index nodes to navigate the tree structure quickly when it wants to find some data (which is always stored in leaf nodes). Index nodes speed a tree search by dividing the tree into smaller pieces, as illustrated in Figure 2-9 (page 2-68).

The records stored in an index node are called **pointer records.** A pointer record consists of a key followed by the node number of the corresponding node. The structure of the key varies according to the type of B*-tree file that contains the index node. For a catalog

file, the search key is a combination of the file or directory name and the parent directory ID of that file or directory. In an extents overflow file, the search key is a combination of that file's type, its file ID, and the index of the first allocation block in the extent. See the sections "Catalog File Keys" on page 2-72 and "Extents Overflow Files" on page 2-75 for more details on the structure of index node search keys.

The immediate descendants of an index node are called the children of the index node. An index node can have from 1 to 15 children, depending on the size of the pointer records that the index node contains. Typically the File Manager selects one of the node's children and continues the search at that node; the File Manager may stop the search, however, if the index node does not contain a pointer record with the appropriate key.

The first index node in a B*-tree is called the **root node.** Recall that the B*-tree header node contains the node number of the root node in the `bthRoot` field of the header record.

Leaf Nodes

The bottom level of a B*-tree structure is occupied exclusively by **leaf nodes,** which contain data records (not pointer records). The structure of the leaf node data records varies according to the type of B*-tree under consideration. In an extents overflow file, the leaf node data records consist of a key and an extent record. In a catalog file (described in the next section), the leaf node data records can be any one of four kinds of records.

Catalog Files

The File Manager uses a file called the **catalog file** to maintain information about the hierarchy of files and directories on a volume. A catalog file is organized as a B*-tree file and hence consists of a header node, index nodes, leaf nodes, and (if necessary) map nodes. The allocation block number of the first file extent of the catalog file (and hence of the file's header node) is stored in the MDB; when the volume is mounted, that information is copied into that volume's volume control block. From the header node, the File Manager can obtain the node number of the catalog file's root node; from the root node, the File Manager can find the entire catalog file.

Each node of the catalog file is assigned a unique **catalog node ID (CNID).** For directories, the CNID is the directory ID; for files, it's the file ID. For any given file or directory, the parent ID is the CNID of the parent directory. The first 16 CNIDs are reserved for use by Apple Computer, Inc., and include the following standard assignments:

CNID	Assignment
1	Parent ID of the root directory
2	Directory ID of the root directory
3	File number of the extents file
4	File number of the catalog file
5	File number of the bad allocation block file

You need to know only two things about a catalog file in addition to the information given earlier in this chapter in "B*-Trees":

- the format of the catalog key used in index and leaf nodes
- the format of the leaf node data records

These formats are described in the following two sections.

Catalog File Keys

The key that the File Manager uses to navigate the catalog file is simple: for a given file or directory, the key consists principally of the name of that file or directory and its parent directory ID. With the exception of a volume reference number (which is not needed here), this mirrors the standard way to specify a file or directory with the high-level HFS routines. You can describe a catalog file key using a record of the `CatKeyRec` data type.

```
TYPE CatKeyRec    =              {catalog key record}
RECORD
   ckrKeyLen:     SignedByte;    {key length}
   ckrResrv1:     SignedByte;    {reserved}
   ckrParID:      LongInt;       {parent directory ID}
   ckrCName:      Str31;         {catalog node name}
END;
```

Field descriptions

`ckrKeyLen` The length (in bytes) of the rest of the key. The value in this field does not include the byte occupied by the field itself. If this field contains 0, the key indicates a deleted record.

`ckrResrv1` Reserved.

`ckrParID` The catalog node ID of the parent directory.

`ckrCName` The name of the file or directory whose catalog entry is to be found. This field is padded with null characters if necessary to have the next record data or pointer begin on a word boundary.

You should pay special attention to the fact that the catalog key differs slightly depending on whether it occurs in a record in an index node or a leaf node. If the key occurs in a pointer record (hence in an index node), the `ckrCName` field always occupies a full 32 bytes and the `ckrKeyLen` field always contains the value $25.

If, however, the catalog file key occurs in a data record (hence in a leaf node), then the `ckrCName` field varies in length; it occupies only the number of bytes required to hold the file or directory name, suitably padded so that the data following it begins on a word boundary. In that case, the `ckrKeyLen` field varies as well and may contain values from $7 to $25.

Catalog File Data Records

A catalog file leaf node can contain four different types of records:

- Directory records. A directory record contains information about a single directory.
- File records. A file record contains information about a single file.
- Directory thread records. A directory thread record provides a link between a directory and its parent directory. It allows the File Manager to find the name and directory ID of the parent of a given directory.
- File thread records. A file thread record provides a link between a file and its parent directory. It allows the File Manager to find the name and directory ID of the parent of a given file.

Each record is defined by a variant of the `CatDataType` data type.

```
TYPE CatDataType  =  (cdrDirRec, cdrFilRec, cdrThdRec,
                      cdrFThdRec);

TYPE CatDataRec   =           {catalog data records}
RECORD
   cdrType:       SignedByte; {record type}
   cdrResrv2:     SignedByte; {reserved}
CASE CatDataType OF
cdrDirRec:                    {directory record}
  (dirFlags:      Integer;    {directory flags}
   dirVal:        Integer;    {directory valence}
   dirDirID:      LongInt;    {directory ID}
   dirCrDat:      LongInt;    {date and time of creation}
   dirMdDat:      LongInt;    {date and time of last modification}
   dirBkDat:      LongInt;    {date and time of last backup}
   dirUsrInfo:    DInfo;      {Finder information}
   dirFndrInfo:   DXInfo;     {additional Finder information}
   dirResrv:      ARRAY[1..4] OF LongInt);
                              {reserved}
cdrFilRec:                    {file record}
  (filFlags:      SignedByte; {file flags}
   filTyp:        SignedByte; {file type}
   filUsrWds:     FInfo;      {Finder information}
   filFlNum:      LongInt;    {file ID}
   filStBlk:      Integer;    {first alloc. blk. of data fork}
   filLgLen:      LongInt;    {logical EOF of data fork}
   filPyLen:      LongInt;    {physical EOF of data fork}
   filRStBlk:     Integer;    {first alloc. blk. of resource fork}
   filRLgLen:     LongInt;    {logical EOF of resource fork}
   filRPyLen:     LongInt;    {physical EOF of resource fork}
   filCrDat:      LongInt;    {date and time of creation}
```

```
   filMdDat:      LongInt;    {date and time of last modification}
   filBkDat:      LongInt;    {date and time of last backup}
   filFndrInfo:   FXInfo;     {additional Finder information}
   filClpSize:    Integer;    {file clump size}
   filExtRec:     ExtDataRec; {first data fork extent record}
   filRExtRec:    ExtDataRec; {first resource fork extent record}
   filResrv:      LongInt);   {reserved}
cdrThdRec:                    {directory thread record}
  (thdResrv:      ARRAY[1..2] OF LongInt;
                              {reserved}
   thdParID:      LongInt;    {parent ID for this directory}
   thdCName:      Str31);     {name of this directory}
cdrFThdRec:                   {file thread record}
  (fthdResrv:     ARRAY[1..2] OF LongInt;
                              {reserved}
   fthdParID:     LongInt;    {parent ID for this file}
   fthdCName:     Str31);     {name of this file}
END;
```

The first two fields of a catalog data record are common to all four variants. Each variant also includes its own unique fields.

Field descriptions common to all variants

`cdrType` The type of catalog data record. This field can contain one of four values:

Value	Meaning
1	Directory record
2	File record
3	Directory thread record
4	File thread record

`cdrResrv2` Reserved.

Field descriptions for the `cdrDirRec` variant

`dirFlags` Directory flags.

`dirVal` The directory valence (the number of files in this directory).

`dirDirID` The directory ID.

`dirCrDat` The date and time this directory was created.

`dirMdDat` The date and time this directory was last modified.

`dirBkDat` The date and time this directory was last backed up.

`dirUsrInfo` Information used by the Finder.

`dirFndrInfo` Additional information used by the Finder.

`dirResrv` Reserved.

Field descriptions for the `cdrFilRec` variant

`filFlags` File flags. This is interpreted as a bitmap; currently the following bits are defined:

Bit	Meaning
0	If set, file is locked and cannot be written to.
1	If set, a file thread record exists for this file.
7	If set, the file record is used.

`filTyp` The file type. This field should always contain 0.

`filUsrWds` The file's Finder information.

`filFlNum` The file ID.

`filStBlk` The first allocation block of the data fork.

`filLgLen` The logical EOF of the data fork.

`filPyLen` The physical EOF of the data fork.

`filRStBlk` The first allocation block of the resource fork.

`filRLgLen` The logical EOF of the resource fork.

`filRPyLen` The physical EOF of the resource fork.

`filCrDat` The date and time this file was created.

`filMdDat` The date and time this file was last modified.

`filBkDat` The date and time this file was last backed up.

`filFndrInfo` Additional information used by the Finder.

`filClpSize` The file clump size.

`filExtRec` The first extent record of the file's data fork.

`filRExtRec` The first extent record of the file's resource fork.

`filResrv` Reserved.

Field descriptions for the `cdrThdRec` variant

`thdResrv` Reserved.

`thdParID` The directory ID of the parent of the associated directory.

`thdCName` The name of this directory.

Field descriptions for the `cdrFThdRec` variant

`fthdResrv` Reserved.

`fthdParID` The directory ID of the parent of the associated file.

`fthdCName` The name of this file.

As you can see, a file thread record is exactly the same as a directory thread record except that the associated object is a file, not a directory.

Extents Overflow Files

The File Manager keeps track of which allocation blocks belong to a file by maintaining a list of contiguous disk segments that belong to that file, in the appropriate order. When the list of disk segments gets too large, some of those segments (or extents) are stored on disk in a file called the **extents overflow file.**

The structure of an extents overflow file is relatively simple compared to that of a catalog file. The function of the extents overflow file is to store those file extents that are not contained in the MDB or VCB (in the case of the catalog and extents overflow files themselves) or in an FCB (in the case of all other files). Because the first three file extents are always maintained in memory (in a VCB or an FCB), the File Manager needs to read the extents overflow file only to retrieve any file extents beyond the first three; if a file has at most three extents, the File Manager never needs to read the disk to find the locations of the file's blocks. (This is one good reason to promote file block contiguity.)

An **extent** is a contiguous range of allocation blocks that have been allocated to some file. You can represent the structure of an extent using an **extent descriptor,** defined by the `ExtDescriptor` data type.

```
TYPE ExtDescriptor      =       {extent descriptor}
RECORD
   xdrStABN:      Integer;       {first allocation block}
   xdrNumABlks:   Integer;       {number of allocation blocks}
END;
```

An extent descriptor record consists of the first allocation block of the extent, followed by the number of allocation blocks in that extent. The File Manager prefers to access extent descriptors in groups of three; to do so, it uses the **extent data record,** defined by the `ExtDataRec` data type.

```
TYPE
   ExtDataRec: ARRAY[1..3] OF ExtDescriptor;{extent data record}
```

Recall that the `drCTExtRec` and `drXTExtRec` fields of the MDB are of type `ExtDataRec` (see "Master Directory Blocks," earlier in this chapter), as is the `fcbExtRec` field of an FCB (see "File Control Blocks" beginning on page 2-82). Also, the records in the leaf nodes of the extents overflow file are extent data records. For this reason, the extents overflow file is much simpler than the catalog file: the data in a leaf node of an extents overflow file always consists of a single kind of record, instead of the four kinds of records found in a catalog file.

The other main difference between a catalog B*-tree and an extents overflow B*-tree concerns the format of the key. You can describe an extent record key with the `ExtKeyRec` data type.

```
TYPE ExtKeyRec    =             {extent key record}
RECORD
   xkrKeyLen:     SignedByte;   {key length}
   xkrFkType:     SignedByte;   {fork type}
   xkrFNum:       LongInt;      {file number}
   xkrFABN:       Integer;      {starting file allocation block}
END;
```

Field descriptions

`xkrKeyLen`	The length (in bytes) of the rest of the key. In the current implementation, this field always contains the value 7.
`xkrFkType`	The type of file fork. This field contains $00 if the file is a data fork and $FF if the file is a resource fork.
`xkrFNum`	The file ID of the file.
`xkrFABN`	The starting file allocation block number. In the list of the allocation blocks belonging to this file, this number is the index of the first allocation block of the first extent descriptor of the extent record.

Note

Disks initialized using the enhanced Disk Initialization Manager introduced in system software version 7.0 might contain extent records for some blocks that do not belong to any actual file in the file system. These extent records have a file ID set to 5, indicating that the extent contains a bad block. See the chapter "Disk Initialization Manager" in this book for details on bad block sparing. ◆

Data Organization in Memory

This section describes the data structures used internally by the File Manager and any external file system that accesses files on Macintosh-initialized volumes. As described in "Data Organization on Volumes," which begins on page 2-53, most applications do not need to access these internal data structures directly. In general, you need to know about these data structures only if you are writing an external file system or a disk utility.

▲ **WARNING**

This section is provided primarily for informational purposes. The organization of data in memory is subject to change. If you want your application to be compatible with future versions of Macintosh system software, you should not access these internal data structures directly. ▲

The data structures maintained in memory by the File Manager and external file systems include

- the file I/O queue
- the volume control block queue, listing information about each mounted volume
- the file control block buffer, listing information about each access path to a fork
- a B*-tree control block for the catalog file and the extents overflow file for each mounted volume
- the drive queue, listing information about each drive connected to the Macintosh

The File I/O Queue

The file I/O queue is a standard Operating System queue (described in the chapter "Queue Utilities" in *Inside Macintosh: Operating System Utilities*) that contains parameter blocks for all asynchronous routines awaiting execution.

Each entry in the file I/O queue consists of a parameter block for the routine that was called. The File Manager uses the first four fields of each parameter block in processing the I/O requests in the queue.

```
TYPE ParamBlockRec =
RECORD
   qLink:         QElemPtr;      {next queue entry}
   qType:         Integer;       {queue type}
   ioTrap:        Integer;       {routine trap}
   ioCmdAddr:     Ptr;           {routine address}
                                 {rest of block}
END;
```

Field descriptions

`qLink`	A pointer to the next entry in the file I/O queue.
`qType`	The queue type. This field must always contain `ORD(ioQType)`.
`ioTrap`	The trap word of the routine that was called.
`ioCmdAddr`	The address of the routine that was called.

You can get a pointer to the header of the file I/O queue by calling the File Manager utility function `GetFSQHdr`.

Assembly-Language Note

The global variable `FSQHdr` contains the header of the file I/O queue. ◆

Volume Control Blocks

Each time a volume is mounted, the File Manager reads its volume information from the master directory block and uses the information to build a new volume control block (VCB) in the volume control block queue (unless an ejected or offline volume is being remounted). The File Manager also creates a volume buffer in the system heap. When a volume is placed offline, its buffer is released. When a volume is unmounted, its VCB is removed from the VCB queue as well.

Assembly-Language Note

The global variable `VCBQHdr` contains the header of the VCB queue. The global variable `DefVCBPtr` points to the VCB of the default volume. ◆

▲ **WARNING**

The size and structure of a VCB may be different in future versions of Macintosh system software. To ensure that you are reading the correct version of a VCB, check the `vcbSigWord` field; it should contain the value $4244. ▲

The **volume control block queue** is a standard Operating System queue that's maintained in the system heap. It contains a volume control block for each mounted volume. A **volume control block** is a nonrelocatable block that contains volume-specific information. The structure of a volume control block is defined by the `VCB` data type.

```
TYPE VCB             =          {volume control block}
RECORD
   qLink:            QElemPtr;  {next queue entry}
   qType:            Integer;   {queue type}
   vcbFlags:         Integer;   {volume flags}
   vcbSigWord:       Integer;   {volume signature}
   vcbCrDate:        LongInt;   {date and time of volume creation}
   vcbLsMod:         LongInt;   {date and time of last modification}
   vcbAtrb:          Integer;   {volume attributes}
   vcbNmFls:         Integer;   {number of files in root directory}
   vcbVBMSt:         Integer;   {first block of volume bitmap}
   vcbAllocPtr:      Integer;   {start of next allocation search}
   vcbNmAlBlks:      Integer;   {number of allocation blocks in volume}
   vcbAlBlkSiz:      LongInt;   {size (in bytes) of allocation blocks}
   vcbClpSiz:        LongInt;   {default clump size}
   vcbAlBlSt:        Integer;   {first allocation block in volume}
   vcbNxtCNID:       LongInt;   {next unused catalog node ID}
   vcbFreeBks:       Integer;   {number of unused allocation blocks}
   vcbVN:            String[27]; {volume name}
   vcbDrvNum:        Integer;   {drive number}
   vcbDRefNum:       Integer;   {driver reference number}
   vcbFSID:          Integer;   {file-system identifier}
   vcbVRefNum:       Integer;   {volume reference number}
   vcbMAdr:          Ptr;       {used internally}
   vcbBufAdr:        Ptr;       {used internally}
   vcbMLen:          Integer;   {used internally}
   vcbDirIndex:      Integer;   {used internally}
   vcbDirBlk:        Integer;   {used internally}
   vcbVolBkUp:       LongInt;   {date and time of last backup}
   vcbVSeqNum:       Integer;   {volume backup sequence number}
   vcbWrCnt:         LongInt;   {volume write count}
   vcbXTClpSiz:      LongInt;   {clump size for extents overflow file}
   vcbCTClpSiz:      LongInt;   {clump size for catalog file}
   vcbNmRtDirs:      Integer;   {number of directories in root dir.}
   vcbFilCnt:        LongInt;   {number of files in volume}
```

```
    vcbDirCnt:        LongInt;    {number of directories in volume}
    vcbFndrInfo:      ARRAY[1..8] OF LongInt;
                                  {information used by the Finder}
    vcbVCSize:        Integer;    {used internally}
    vcbVBMCSiz:       Integer;    {used internally}
    vcbCtlCSiz:       Integer;    {used internally}
    vcbXTAlBks:       Integer;    {size of extents overflow file}
    vcbCTAlBks:       Integer;    {size of catalog file}
    vcbXTRef:         Integer;    {ref. num. for extents overflow file}
    vcbCTRef:         Integer;    {ref. num. for catalog file}
    vcbCtlBuf:        Ptr;        {ptr. to extents and catalog caches}
    vcbDirIDM:        LongInt;    {directory last searched}
    vcbOffsM:         Integer;    {offspring index at last search}
END;
```

Note

The values in the `vcbNmAlBlks` and `vcbFreeBks` fields are unsigned integers (that is, they can range from 0 to 65,535, not from –32,768 to 32,767). Because Pascal does not support unsigned data types, you need to use the technique illustrated in "Determining the Amount of Free Space on a Volume" on page 2-47 to read the values in these fields correctly. ◆

Field descriptions

`qLink` A pointer to the next entry in the VCB queue. You can get a pointer to the header of the VCB queue by calling the File Manager utility function `GetVCBQHdr`.

`qType` The queue type. When the volume is mounted and the VCB is created, this field is cleared. Thereafter, bit 7 of this field is set whenever a file on that volume is opened.

`vcbFlags` Volume flags. Bit 15 is set if the volume information has been changed by a File Manager call since the volume was last affected by a `FlushVol` call.

`vcbSigWord` The volume signature. For HFS volumes, this field contains $4244.

`vcbCrDate` The date and time of volume creation (initialization).

`vcbLsMod` The date and time of last modification. This is not necessarily when the volume was last flushed.

`vcbAtrb` Volume attributes. The bits have these meanings:

Bit	Meaning
0–5	Reserved
6	Set if the volume is busy (one or more files are open)
7	Set if the volume is locked by hardware
8–14	Reserved
15	Set if the volume is locked by software

`vcbNmFls` The number of files in the root directory.

vcbVBMSt	The first block of the volume bitmap.
vcbAllocPtr	The start block of the next allocation search. Used internally.
vcbNmAlBlks	The number of allocation blocks in the volume.
vcbAlBlkSiz	The allocation block size (in bytes). This value must always be a multiple of 512 bytes.
vcbClpSiz	The default clump size.
vcbAlBlSt	The first allocation block in the volume.
vcbNxtCNID	The next unused catalog node ID (directory ID or file ID).
vcbFreeBks	The number of unused allocation blocks on the volume.
vcbVN	The volume name. This field consists of a length byte followed by 27 bytes. Note that the volume name can occupy at most 27 characters; this is an exception to the normal file and directory name limit of 31 characters.
vcbDrvNum	The drive number of the drive on which the volume is located. When a mounted volume is placed offline or ejected, vcbDrvNum is set to 0.
vcbDRefNum	The driver reference number of the driver used to access the volume. When a volume is ejected, vcbDRefNum is set to the previous value of vcbDrvNum (and hence is a positive number). When a volume is placed offline, vcbDRefNum is set to the negative of the previous value of vcbDrvNum (and hence is a negative number).
vcbFSID	An identifier for the file system handling the volume; it's zero for volumes handled by the File Manager and nonzero for volumes handled by other file systems.
vcbVRefNum	The volume reference number.
vcbMAdr	Used internally.
vcbBufAdr	Used internally.
vcbMLen	Used internally.
vcbDirIndex	Used internally.
vcbDirBlk	Used internally.
vcbVolBkUp	The date and time of the last volume backup.
vcbVSeqNum	Used internally.
vcbWrCnt	The volume write count.
vcbXTClpSiz	The clump size of the extents overflow file.
vcbCTClpSiz	The clump size of the catalog file.
vcbNmRtDirs	The number of directories in the root directory.
vcbFilCnt	The number of files on the volume.
vcbDirCnt	The number of directories on the volume.
vcbFndrInfo	Information used by the Finder.
vcbVCSize	Used internally.
vcbVBMCSiz	Used internally.
vcbCtlCSiz	Used internally.

vcbXTAlBks	The size (in blocks) of the extents overflow file.
vcbCTAlBks	The size (in blocks) of the catalog file.
vcbXTRef	The path reference number for the extents overflow file.
vcbCTRef	The path reference number for the catalog file.
vcbCtlBuf	A pointer to the extents and catalog caches.
vcbDirIDM	The directory last searched.
vcbOffsM	The offspring index at the last search.

File Control Blocks

Each time a file is opened, the File Manager reads that file's catalog entry and builds a **file control block** (FCB) in the FCB buffer, which contains information about all access paths. The FCB buffer is a block in the system heap; the first word contains the length of the buffer, and the remainder of the buffer is used to hold FCBs for open files.

The initial size of the FCB buffer is determined by the system startup information stored on a volume. Beginning in system software version 7.0, the File Manager attempts to resize the FCB buffer whenever the existing buffer is filled.

You can find the beginning of any particular FCB by adding the size of all preceding FCBs to the size of the FCB buffer length word (that is, 2). This offset from the head of the FCB buffer is used as the file reference number of the corresponding open file. Because the current size of an FCB is 94 bytes, the first few valid file reference numbers are 2, 96, 190, 284, 378, 472, and so on. The maximum size of an expandable FCB buffer is 32,535 bytes, so there is an absolute limit of 342 FCBs in the FCB buffer.

Note
The size and structure of an FCB will be different in future versions of Macintosh system software. To be safe, you should get information from the FCB allocated for an open file by calling the File Manager function `PBGetFCBInfo`. ◆

When you close a file (for example, by calling `FSClose`), the FCB for that file is cleared, and the File Manager may use that space to hold the FCB for a file that is opened at a later time. Consequently, it is important that you do not attempt to close a file more than once; you may inadvertently close a file that was opened by the system or by another application.

▲ **WARNING**
Closing a volume's catalog file (perhaps by inadvertently calling `FSClose` or `PBClose` twice with the same file reference number) may result in damage to the volume's file system and loss of data. ▲

The structure of a file control block is defined by the `FCB` data type.

```
TYPE FCB    =                {file control block}
RECORD
   fcbFlNum:      LongInt;    {file ID}
   fcbFlags:      Integer;    {file flags}
```

```
   fcbSBlk:       Integer;    {reserved}
   fcbEOF:        LongInt;    {logical end-of-file}
   fcbPLen:       LongInt;    {physical end-of-file}
   fcbCrPs:       LongInt;    {current file mark position}
   fcbVPtr:       Ptr;        {pointer to volume control block}
   fcbBfAdr:      Ptr;        {pointer to access path buffer}
   fcbFlPos:      Integer;    {reserved}
   fcbClmpSize:   LongInt;    {file clump size}
   fcbBTCBPtr:    Ptr;        {pointer to B*-tree control block}
   fcbExtRec:     ExtDataRec; {first three file extents}
   fcbFType:      LongInt;    {file's four Finder type bytes}
   fcbCatPos:     LongInt;    {catalog hint for use on close}
   fcbDirID:      LongInt;    {file's parent directory ID}
   fcbCName:      String[31]; {name of file}
END;
```

Field descriptions

`fcbFlNum` The file ID of this file.

`fcbFlags` Flags describing the status of the file. Currently the following bits are defined:

Bit	Meaning
0–7	Reserved
8	Set if data can be written to the file
9	Set if this FCB describes a resource fork
10	Set if the file has a locked byte range
11	Reserved
12	Set if the file has shared write permissions
13	Set if the file is locked (write-protected)
14	Set if the file's clump size is specified in the FCB
15	Set if the file has changed since it was last flushed

`fcbSBlk` Reserved.

`fcbEOF` The logical end-of-file of the file.

`fcbPLen` The physical end-of-file of the file.

`fcbCrPs` The position of the mark.

`fcbVPtr` A pointer to the volume control block of the volume containing the file.

`fcbBfAdr` A pointer to the file's access path buffer.

`fcbFlPos` Reserved.

`fcbClmpSize` The clump size of the file.

`fcbBTCBPtr` A pointer to the file's B*-tree control block.

`fcbExtRec` An extent record (12 bytes) containing the first three extents of the file.

`fcbFType`	The file's Finder type.
`fcbCatPos`	A catalog hint, used when you close the file.
`fcbDirID`	The file's parent directory ID.
`fcbCName`	The file's name (as contained in the volume catalog file).

B*-Tree Control Blocks

When the File Manager mounts a volume, it reads the B*-tree header node for both the catalog file and the extents overflow file found on that volume and, for each file, creates a **B*-tree control block** in memory. (See the section "Header Nodes" on page 2-68 for a description of B*-tree header nodes.) The structure of a B*-tree control block is defined by the `BTCB` data type.

```
TYPE BTCB         =           {B*-tree control block}
RECORD
   btcFlags:      SignedByte; {flag byte}
   btcResv:       SignedByte; {reserved}
   btcRefNum:     Integer;    {file reference number}
   btcKeyCr:      ProcPtr:    {pointer to key comparison routine}
   btcCQPtr:      LongInt;    {pointer to cache queue}
   btcVarPtr:     LongInt;    {pointer to B*-tree variables}
   btcLevel:      Integer;    {current level}
   btcNodeM:      LongInt;    {current node mark}
   btcIndexM:     Integer;    {current index mark}
   btcDepth:      Integer;    {current depth of tree}
   btcRoot:       LongInt;    {number of root node}
   btcNRecs:      LongInt;    {number of leaf records in tree}
   btcFNode:      LongInt;    {number of first leaf node}
   btcLNode:      LongInt;    {number of last leaf node}
   btcNodeSize:   Integer;    {size of a node}
   btcKeyLen:     Integer;    {maximum length of a key}
   btcNNodes:     LongInt;    {total number of nodes in tree}
   btcFree:       LongInt;    {number of free nodes}
END;
```

Field descriptions

`btcFlags` A flag byte. Currently the following bits are defined:

Bit	Meaning
4	Set if an existing index record must be deleted
5	Set if a new index record must be created
6	Set if the index key must be updated
7	Set if the block has changed since it was last flushed

btcResv	Reserved.
btcRefNum	The file reference number of the catalog or extents overflow file corresponding to this control block.
btcKeyCr	A pointer to the routine used to compare keys.
btcCQPtr	A pointer to the cache queue.
btcVarPtr	A pointer to B*-tree variables.
btcLevel	The current level.
btcNodeM	The current node mark.
btcIndexM	The current index mark.
bthDepth	The current depth of the B*-tree.
btcRoot	The node number of the root node. The root node is the start of the B*-tree structure; usually the root node is the first index node, but it might be a leaf node if there are no index nodes.
btcNRecs	The number of data records (records contained in leaf nodes).
btcFNode	The node number of the first leaf node.
btcLNode	The node number of the last leaf node.
btcNodeSize	The size (in bytes) of a node. Currently, this is always 512.
btcKeyLen	The length of the key records in each node.
btcNNodes	The total number of nodes in the B*-tree.
btcFree	The total number of free nodes in the B*-tree.

The Drive Queue

The File Manager maintains a list of all disk drives connected to the computer. It maintains this list in the **drive queue,** which is a standard operating system queue. The drive queue is initially created at system startup time. Elements are added to the queue at system startup time or when you call the AddDrive procedure. The drive queue can support any number of drives, limited only by memory space. Each element in the drive queue contains information about the corresponding drive; the structure of a drive queue element is defined by the DrvQEl data type.

```
TYPE DrvQEl =
RECORD
   qLink:         QElemPtr;   {next queue entry}
   qType:         Integer;    {flag for dQDrvSz and dQDrvSz2}
   dQDrive:       Integer;    {drive number}
   dQRefNum:      Integer;    {driver reference number}
   dQFSID:        Integer;    {file-system identifier}
   dQDrvSz:       Integer;    {number of logical blocks on drive}
   dQDrvSz2:      Integer;    {additional field for large drives}
END;
```

Field descriptions

`qLink`	A pointer to the next entry in the drive queue.
`qType`	Used to specify the size of the drive. If the value of `qType` is 0, the number of logical blocks on the drive is contained in the `dQDrvSz` field alone. If the value of `qType` is 1, both `dQDrvSz` and `dQDrvSz2` are used to store the number of blocks; in that case, `dQDrvSz2` contains the high-order word of this number and `dQDrvSz` contains the low-order word.
`dQDrive`	The drive number of the drive.
`dQRefNum`	The driver reference number of the driver controlling the device on which the volume is mounted.
`dQFSID`	An identifier for the file system handling the volume in the drive; it's zero for volumes handled by the File Manager and nonzero for volumes handled by other file systems.
`dQDrvSz`	The number of logical blocks on the drive.
`dQDrvSz2`	An additional field to handle large drives. This field is used only if the `qType` field contains 1.

The File Manager also maintains four flag bytes preceding each drive queue element. These bytes contain the following information:

Byte	Contents
0	Bit 7=1 if the volume on the drive is locked
1	0 if no disk in drive; 1 or 2 if disk in drive; 8 if nonejectable disk in drive; $FC–$FF if disk was ejected within last 1.5 seconds; $48 if disk in drive is nonejectable but driver wants a call
2	Used internally during system startup
3	Bit 7=0 if disk is single-sided

You can read these flags by subtracting 4 bytes from the beginning of a drive queue element, as illustrated in Listing 2-11.

Listing 2-11 Reading a drive queue element's flag bytes

```
FUNCTION GetDriveFlags (myDQElemPtr: DrvQElPtr): LongInt;
TYPE
   FlagPtr = ^LongInt;  {pointer to the queue element flag bytes}
VAR
   myQFlagsPtr:   FlagPtr;
BEGIN
   {Just subtract 4 from the queue element pointer.}
   myQFlagsPtr := FlagPtr(ORD4(myDQElemPtr) - 4);
   GetDriveFlags := myQFlagsPtr^;
END;
```

The `GetDriveFlags` function defined Listing 2-11 takes a pointer to a drive queue element as a parameter. You can get a queue element pointer for a particular volume by walking the drive queue until you find a queue element whose `dQDrive` field contains the same value as the `vcbDrvNum` field of that volume's VCB. You can get a pointer to the header of the drive queue by calling the File Manager function `GetDrvQHdr`.

Note that the bit numbers given in this section use the standard MC68000 numbering scheme; to access the correct bit using some Pascal routines, you must reverse that numbering. For example, if you use the Toolbox `BitTst` routine to determine whether a particular disk is single-sided, you must test bit 24 (that is, 31 minus 7) of the returned long integer. If you use the built-in Pascal function `BTST`, however, you can test the indicated bit directly.

Assembly-Language Note

The global variable `DrvQHdr` contains the header of the drive queue. ◆

File Manager Reference

This section describes the routines provided by the File Manager and the data structures you must pass when calling those routines.

The "Data Structures" section shows the Pascal data structures for all the records and parameter blocks that most applications are likely to use. If you need information about data structures describing the structure of the information maintained on volumes or in memory, see "Data Organization on Volumes" and "Data Organization in Memory" earlier in this chapter.

The remaining sections describe the routines provided by the File Manager.

Data Structures

This section describes the data structures that your application uses to exchange information with the File Manager.

File System Specification Record

The system software recognizes the file system specification record, which provides a simple, standard way to specify the name and location of a file or directory. The file system specification record is defined by the `FSSpec` data type.

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

Field descriptions

`vRefNum`	The volume reference number of the volume containing the specified file or directory.
`parID`	The directory ID of the directory containing the specified file or directory.
`name`	The name of the specified file or directory.

The `FSSpec` record can describe only a file or a directory, not a volume. A volume can be identified by its root directory, although the system software never uses an `FSSpec` record to describe a volume. (The directory ID of the root's parent directory is `fsRtParID`, defined in the interface files. The name of the root directory is the same as the name of the volume.)

If you need to convert a file specification into an `FSSpec` record, call the function `FSMakeFSSpec`. Do not fill in the fields of an `FSSpec` record yourself.

Basic File Manager Parameter Block

Many of the low-level functions that manipulate files and volumes exchange information with your application using the basic File Manager parameter block, defined by the `ParamBlockRec` data type.

```
TYPE ParamBlockRec   =                {basic File Manager parameter block}
RECORD
      qLink:            QElemPtr;     {next queue entry}
      qType:            Integer;      {queue type}
      ioTrap:           Integer;      {routine trap}
      ioCmdAddr:        Ptr;          {routine address}
      ioCompletion:     ProcPtr;      {pointer to completion routine}
      ioResult:         OSErr;        {result code}
      ioNamePtr:        StringPtr;    {pointer to pathname}
      ioVRefNum:        Integer;      {volume specification}
   CASE ParamBlkType OF
   ioParam:
     (ioRefNum:         Integer;      {file reference number}
      ioVersNum:        SignedByte;   {version number}
      ioPermssn:        SignedByte;   {read/write permission}
      ioMisc:           Ptr;          {miscellaneous}
      ioBuffer:         Ptr;          {data buffer}
      ioReqCount:       LongInt;      {requested number of bytes}
      ioActCount:       LongInt;      {actual number of bytes}
      ioPosMode:        Integer;      {positioning mode and newline char.}
      ioPosOffset:      LongInt);     {positioning offset}
   fileParam:
     (ioFRefNum:        Integer;      {file reference number}
      ioFVersNum:       SignedByte;   {file version number (unused)}
```

```
    filler1:          SignedByte;    {reserved}
    ioFDirIndex:      Integer;       {directory index}
    ioFlAttrib:       SignedByte;    {file attributes}
    ioFlVersNum:      SignedByte;    {file version number (unused)}
    ioFlFndrInfo:     FInfo;         {information used by the Finder}
    ioFlNum:          LongInt;       {file ID}
    ioFlStBlk:        Integer;       {first alloc. blk. of data fork}
    ioFlLgLen:        LongInt;       {logical EOF of data fork}
    ioFlPyLen:        LongInt;       {physical EOF of data fork}
    ioFlRStBlk:       Integer;       {first alloc. blk. of resource fork}
    ioFlRLgLen:       LongInt;       {logical EOF of resource fork}
    ioFlRPyLen:       LongInt;       {physical EOF of resource fork}
    ioFlCrDat:        LongInt;       {date and time of creation}
    ioFlMdDat:        LongInt);      {date and time of last modification}
  volumeParam:
    (filler2:         LongInt;       {reserved}
    ioVolIndex:       Integer;       {volume index}
    ioVCrDate:        LongInt;       {date and time of initialization}
    ioVLsBkUp:        LongInt;       {date and time of last modification}
    ioVAtrb:          Integer;       {volume attributes}
    ioVNmFls:         Integer;       {number of files in root directory}
    ioVDirSt:         Integer;       {first block of directory}
    ioVBlLn:          Integer;       {length of directory in blocks}
    ioVNmAlBlks:      Integer;       {number of allocation blocks}
    ioVAlBlkSiz:      LongInt;       {size of allocation blocks}
    ioVClpSiz:        LongInt;       {default clump size}
    ioAlBlSt:         Integer;       {first block in block map}
    ioVNxtFNum:       LongInt;       {next unused file ID}
    ioVFrBlk:         Integer);      {number of unused allocation blocks}
  END;
```

The first eight fields are common to all three variants. Each variant also includes its own unique fields.

Field descriptions for fields common to all variants

`qLink` A pointer to the next entry in the file I/O queue. (This field is used internally by the File Manager to keep track of asynchronous calls awaiting execution.)

`qType` The queue type. (This field is used internally by the File Manager.)

`ioTrap` The trap number of the routine that was called. (This field is used internally by the File Manager.)

`ioCmdAddr` The address of the routine that was called. (This field is used internally by the File Manager.)

`ioCompletion` A pointer to a completion routine to be executed at the end of an asynchronous call. It should be `NIL` for asynchronous calls with no completion routine and is automatically set to `NIL` for all synchronous calls. See "Completion Routines" on page 2-240 for information about completion routines.

`ioResult` The result code of the function. For synchronous calls, this field is the same as the result code of the function call itself. To determine when an asynchronous call has actually been completed, your application can poll this field; it's set to a positive number when the call is made and receives the actual result code when the call is completed.

`ioNamePtr` A pointer to a pathname. Whenever a routine description specifies that `ioNamePtr` is used—whether for input, output, or both—it's very important that you set this field to point to storage for a `Str255` value (if you're using a pathname) or to `NIL` (if you're not).

`ioVRefNum` A volume specification (volume reference number, working directory reference number, drive number, or 0 for default volume).

Field descriptions for the `ioParam` variant

`ioRefNum` The file reference number of an open file.

`ioVersNum` A version number. This field is no longer used and you should always set it to 0.

`ioPermssn` The access mode.

`ioMisc` Depends on the routine called. This field contains either a new logical end-of-file, a new version number, or a pointer to a new pathname. Because `ioMisc` is of type `Ptr`, you'll need to perform type coercion to interpret the value of `ioMisc` correctly when it contains an end-of-file (a `LongInt` value) or version number (a `SignedByte` value).

`ioBuffer` A pointer to a data buffer into which data is written by `_Read` calls and from which data is read by `_Write` calls.

`ioReqCount` The requested number of bytes to be read, written, or allocated.

`ioActCount` The number of bytes actually read, written, or allocated.

`ioPosMode` The positioning mode for setting the mark. Bits 0 and 1 of this field indicate how to position the mark; you can use the following predefined constants to set or test their value:

```
CONST
   fsAtMark       =  0; {at current mark}
   fsFromStart    =  1; {from beginning of file}
   fsFromLEOF     =  2; {from logical end-of-file}
   fsFromMark     =  3; {relative to current mark}
```

You can set bit 4 of the `ioPosMode` field to request that the data be cached, and you can set bit 5 to request that the data not be cached. You can set bit 6 to request that any data written be immediately

read; this ensures that the data written to a volume exactly matches the data in memory. To request a read-verify operation, add the following constant to the positioning mode:

```
CONST
   rdVerify       =  64;   {use read-verify mode}
```

You can set bit 7 to read a continuous stream of bytes, and place the ASCII code of a newline character in the high-order byte to terminate a read operation at the end of a line.

`ioPosOffset` The offset to be used in conjunction with the positioning mode.

Field descriptions for the `fileParam` variant

`ioFRefNum` The file reference number of an open file.

`ioFVersNum` A file version number. This field is no longer used and you should always set it to 0.

`filler1` Reserved.

`ioFDirIndex` An index for use with the `PBHGetFInfo` function.

`ioFlAttrib` File attributes. The bits in this field have these meanings:

Bit	Meaning
0	Set if file is locked
2	Set if resource fork is open
3	Set if data fork is open
4	Set if a directory
7	Set if file (either fork) is open

`ioFlVersNum` A file version number. This feature is no longer supported, and you must always set this field to 0.

`ioFlFndrInfo` Information used by the Finder. (See the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials* for details.)

`ioFlNum` A file ID.

`ioFlStBlk` The first allocation block of the data fork. This field contains 0 if the file’s data fork is empty.

`ioFlLgLen` The logical end-of-file of the data fork.

`ioFlPyLen` The physical end-of-file of the data fork.

`ioFlRStBlk` The first allocation block of the resource fork. This field contains 0 if the file’s resource fork is empty.

`ioFlRLgLen` The logical end-of-file of the resource fork.

`ioFlRPyLen` The physical end-of-file of the resource fork.

`ioFlCrDat` The date and time of the file’s creation, specified in seconds since midnight, January 1, 1904.

`ioFlMdDat` The date and time of the last modification to the file, specified in seconds since midnight, January 1, 1904.

Field descriptions for the `volumeParam` variant

`filler2`	Reserved.
`ioVolIndex`	The volume index.
`ioVCrDate`	The date and time of volume initialization.
`ioVLsBkUp`	The date and time the volume information was last modified. (This field is not changed when information is written to a file and does not necessarily indicate when the volume was flushed.)
`ioVAtrb`	The volume attributes.
`ioVNmFls`	The number of files in the root directory.
`ioVDirSt`	The first block of the volume directory.
`ioVBlLn`	Length of directory in blocks.
`ioVNmAlBlks`	The number of allocation blocks.
`ioVAlBlkSiz`	The size of allocation blocks.
`ioVClpSiz`	The volume clump size.
`ioAlBlSt`	The first block in the volume map.
`ioVNxtFNum`	The next unused file number.
`ioVFrBlk`	The number of unused allocation blocks.

HFS Parameter Block

Most of the low-level HFS functions exchange information with your application using the HFS parameter block, defined by the `HParamBlockRec` data type.

```
TYPE HParamBlockRec     =               {HFS parameter block}
   RECORD
      qLink:            QElemPtr;       {next queue entry}
      qType:            Integer;        {queue type}
      ioTrap:           Integer;        {routine trap}
      ioCmdAddr:        Ptr;            {routine address}
      ioCompletion:     ProcPtr;        {pointer to completion routine}
      ioResult:         OSErr;          {result code}
      ioNamePtr:        StringPtr;      {pointer to pathname}
      ioVRefNum:        Integer;        {volume specification}
   CASE ParamBlkType OF
   ioParam:
     (ioRefNum:         Integer;        {file reference number}
      ioVersNum:        SignedByte;     {version number}
      ioPermssn:        SignedByte;     {read/write permission}
      ioMisc:           Ptr;            {miscellaneous}
      ioBuffer:         Ptr;            {data buffer}
      ioReqCount:       LongInt;        {requested number of bytes}
      ioActCount:       LongInt;        {actual number of bytes}
      ioPosMode:        Integer;        {positioning mode and newline char.}
      ioPosOffset:      LongInt);       {positioning offset}
```

```
fileParam:
  (ioFRefNum:        Integer;       {file reference number}
   ioFVersNum:       SignedByte;    {file version number (unused)}
   filler1:          SignedByte;    {reserved}
   ioFDirIndex:      Integer;       {directory index}
   ioFlAttrib:       SignedByte;    {file attributes}
   ioFlVersNum:      SignedByte;    {file version number (unused)}
   ioFlFndrInfo:     FInfo;         {information used by the Finder}
   ioDirID:          LongInt;       {directory ID or file ID}
   ioFlStBlk:        Integer;       {first alloc. blk. of data fork}
   ioFlLgLen:        LongInt;       {logical EOF of data fork}
   ioFlPyLen:        LongInt;       {physical EOF of data fork}
   ioFlRStBlk:       Integer;       {first alloc. blk. of resource fork}
   ioFlRLgLen:       LongInt;       {logical EOF of resource fork}
   ioFlRPyLen:       LongInt;       {physical EOF of resource fork}
   ioFlCrDat:        LongInt;       {date and time of creation}
   ioFlMdDat:        LongInt);      {date and time of last modification}
volumeParam:
  (filler2:          LongInt;       {reserved}
   ioVolIndex:       Integer;       {volume index}
   ioVCrDate:        LongInt;       {date and time of initialization}
   ioVLsMod:         LongInt;       {date and time of last modification}
   ioVAtrb:          Integer;       {volume attributes}
   ioVNmFls:         Integer;       {number of files in root directory}
   ioVBitMap:        Integer;       {first block of volume bitmap}
   ioAllocPtr:       Integer;       {first block of next new file}
   ioVNmAlBlks:      Integer;       {number of allocation blocks}
   ioVAlBlkSiz:      LongInt;       {size of allocation blocks}
   ioVClpSiz:        LongInt;       {default clump size}
   ioAlBlSt:         Integer;       {first block in volume map}
   ioVNxtCNID:       LongInt;       {next unused node ID}
   ioVFrBlk:         Integer;       {number of unused allocation blocks}
   ioVSigWord:       Integer;       {volume signature}
   ioVDrvInfo:       Integer;       {drive number}
   ioVDRefNum:       Integer;       {driver reference number}
   ioVFSID:          Integer;       {file-system identifier}
   ioVBkUp:          LongInt;       {date and time of last backup}
   ioVSeqNum:        Integer;       {used internally}
   ioVWrCnt:         LongInt;       {volume write count}
   ioVFilCnt:        LongInt;       {number of files on volume}
   ioVDirCnt:        LongInt;       {number of directories on volume}
   ioVFndrInfo:      ARRAY[1..8] OF LongInt);
                                    {information used by the Finder}
accessParam:
  (filler3:          Integer;       {reserved}
```

```
    ioDenyModes:      Integer;       {access mode information}
    filler4:          Integer;       {reserved}
    filler5:          SignedByte;    {reserved}
    ioACUser:         SignedByte;    {user access rights}
    filler6:          LongInt;       {reserved}
    ioACOwnerID:      LongInt;       {owner ID}
    ioACGroupID:      LongInt;       {group ID}
    ioACAccess:       LongInt);      {directory access rights}
objParam:
  (filler7:           Integer;       {reserved}
    ioObjType:        Integer;       {function code}
    ioObjNamePtr:     Ptr;           {ptr to returned creator/group name}
    ioObjID:          LongInt);      {creator/group ID}
copyParam:
  (ioDstVRefNum:      Integer;       {destination volume identifier}
    filler8:          Integer;       {reserved}
    ioNewName:        Ptr;           {pointer to destination pathname}
    ioCopyName:       Ptr;           {pointer to optional name}
    ioNewDirID:       LongInt);      {destination directory ID}
wdParam:
  (filler9:           Integer;       {reserved}
    ioWDIndex:        Integer;       {working directory index}
    ioWDProcID:       LongInt;       {working directory user identifier}
    ioWDVRefNum:      Integer;       {working directory's vol. ref. num.}
    filler10:         Integer;       {reserved}
    filler11:         LongInt;       {reserved}
    filler12:         LongInt;       {reserved}
    filler13:         LongInt;       {reserved}
    ioWDDirID:        LongInt);      {working directory's directory ID}
fidParam:
  (filler14:          LongInt;       {reserved}
    ioDestNamePtr:    StringPtr;     {pointer to destination filename}
    filler15:         LongInt;       {reserved}
    ioDestDirID:      LongInt;       {destination parent directory ID}
    filler16:         LongInt;       {reserved}
    filler17:         LongInt;       {reserved}
    ioSrcDirID:       LongInt;       {source parent directory ID}
    filler18:         Integer;       {reserved}
    ioFileID:         LongInt);      {file ID}
csParam:
  (ioMatchPtr:        FSSpecArrayPtr;{pointer to array of matches}
    ioReqMatchCount:  LongInt;       {max. number of matches to return}
    ioActMatchCount:  LongInt;       {actual number of matches}
    ioSearchBits:     LongInt;       {enable bits for matching rules}
    ioSearchInfo1:    CInfoPBPtr;    {pointer to values and lower bounds}
```

```
    ioSearchInfo2:    CInfoPBPtr;    {pointer to masks and upper bounds}
    ioSearchTime:     LongInt;       {maximum time to search}
    ioCatPosition:    CatPositionRec;{current catalog position}
    ioOptBuffer:      Ptr;           {pointer to optional read buffer}
    ioOptBufSize:     LongInt);      {length of optional read buffer}
foreignPrivParam:
  (filler21:               LongInt;    {reserved}
   filler22:               LongInt;    {reserved}
   ioForeignPrivBuffer:    Ptr;        {privileges data buffer}
   ioForeignPrivReqCount:  LongInt;    {size of buffer}
   ioForeignPrivActCount:  LongInt;    {amount of buffer used}
   filler23:               LongInt;    {reserved}
   ioForeignPrivDirID:     LongInt;    {parent directory ID of }
                                       { foreign file or directory}
   ioForeignPrivInfo1:     LongInt;    {privileges data}
   ioForeignPrivInfo2:     LongInt;    {privileges data}
   ioForeignPrivInfo3:     LongInt;    {privileges data}
   ioForeignPrivInfo4:     LongInt);   {privileges data}
END;
```

The first eight fields are common to all ten variants. Each variant also includes its own unique fields.

Field descriptions common to all variants

`qLink` A pointer to the next entry in the file I/O queue. (This field is used internally by the File Manager to keep track of asynchronous calls awaiting execution.)

`qType` The queue type. (This field is used internally by the File Manager.)

`ioTrap` The trap number of the routine that was called. (This field is used internally by the File Manager.)

`ioCmdAddr` The address of the routine that was called. (This field is used internally by the File Manager.)

`ioCompletion` A pointer to a completion routine to be executed at the end of an asynchronous call. It should be `NIL` for asynchronous calls with no completion routine and is automatically set to `NIL` for all synchronous calls. See "Completion Routines" on page 2-240 for information about completion routines.

`ioResult` The result code of the function. For synchronous calls, this field is the same as the result code of the function call itself. To determine when an asynchronous call has actually been completed, your application can poll this field; it's set to a positive number when the call is made and receives the actual result code when the call is completed.

`ioNamePtr` A pointer to a pathname. Whenever a routine description specifies that `ioNamePtr` is used—whether for input, output, or both—it's very important that you set this field to point to storage for a `Str255` value (if you're using a pathname) or to `NIL` (if you're not).

`ioVRefNum` A volume specification (volume reference number, working directory reference number, drive number, or 0 for default volume).

Field descriptions for the `ioParam` variant

`ioRefNum` The file reference number of an open file.

`ioVersNum` A version number. This field is no longer used and you should always set it to 0.

`ioPermssn` The access mode.

`ioMisc` Depends on the routine called. This field contains either a new logical end-of-file, a new version number, a pointer to an access path buffer, or a pointer to a new pathname. Because `ioMisc` is of type `Ptr`, you'll need to perform type coercion to interpret the value of `ioMisc` correctly when it contains an end-of-file (a `LongInt` value) or version number (a `SignedByte` value).

`ioBuffer` A pointer to a data buffer into which data is written by `_Read` calls and from which data is read by `_Write` calls.

`ioReqCount` The requested number of bytes to be read, written, or allocated.

`ioActCount` The number of bytes actually read, written, or allocated.

`ioPosMode` The positioning mode for setting the mark. Bits 0 and 1 of this field indicate how to position the mark; you can use the following predefined constants to set or test their value:

```
CONST
   fsAtMark       =  0; {at current mark}
   fsFromStart    =  1; {from beginning of file}
   fsFromLEOF     =  2; {from logical end-of-file}
   fsFromMark     =  3; {relative to current mark}
```

You can set bit 4 of the `ioPosMode` field to request that the data be cached, and you can set bit 5 to request that the data not be cached. You can set bit 6 to request that any data written be immediately read; this ensures that the data written to a volume exactly matches the data in memory. To request a read-verify operation, add the following constant to the positioning mode:

```
CONST
   rdVerify       =  64;   {use read-verify mode}
```

You can set bit 7 to read a continuous stream of bytes, and place the ASCII code of a newline character in the high-order byte to terminate a read operation at the end of a line.

`ioPosOffset` The offset to be used in conjunction with the positioning mode.

Field descriptions for the `fileParam` variant

`ioFRefNum` The file reference number of an open file.

`ioFVersNum` A file version number. This field is no longer used and you should always set it to 0.

`filler1` Reserved.

`ioFDirIndex` An index for use with the `PBHGetFInfo` function.

`ioFlAttrib` File attributes. The bits in this field have these meanings:

Bit	Meaning
0	Set if file is locked
2	Set if resource fork is open
3	Set if data fork is open
4	Set if a directory
7	Set if file (either fork) is open

`ioFlVersNum` A file version number. This field is no longer used and you should always set it to 0.

`ioFlFndrInfo` Information used by the Finder.

`ioDirID` A directory ID.

`ioFlStBlk` The first allocation block of the data fork. This field contains 0 if the file's data fork is empty.

`ioFlLgLen` The logical end-of-file of the data fork.

`ioFlPyLen` The physical end-of-file of the data fork.

`ioFlRStBlk` The first allocation block of the resource fork.

`ioFlRLgLen` The logical end-of-file of the resource fork.

`ioFlRPyLen` The physical end-of-file of the resource fork.

`ioFlCrDat` The date and time of the file's creation, specified in seconds since midnight, January 1, 1904.

`ioFlMdDat` The date and time of the last modification to the file, specified in seconds since midnight, January 1, 1904.

Field descriptions for the `volumeParam` variant

`filler2` Reserved.

`ioVolIndex` An index for use with the `PBHGetVInfo` function.

`ioVCrDate` The date and time of volume initialization.

`ioVLsMod` The date and time the volume information was last modified. (This field is not changed when information is written to a file and does not necessarily indicate when the volume was flushed.)

`ioVAtrb` The volume attributes.

`ioVNmFls` The number of files in the root directory.

`ioVBitMap` The first block of the volume bitmap.

`ioAllocPtr` The block at which the next new file starts. Used internally.

`ioVNmAlBlks` The number of allocation blocks.

`ioVAlBlkSiz` The size of allocation blocks.

`ioVClpSiz` The clump size.

`ioAlBlSt` The first block in the volume map.

`ioVNxtCNID` The next unused catalog node ID.

`ioVFrBlk` The number of unused allocation blocks.

ioVSigWord	A signature word identifying the type of volume; it's $D2D7 for MFS volumes and $4244 for volumes that support HFS calls.
ioVDrvInfo	The drive number of the drive containing the volume.
ioVDRefNum	For online volumes, the reference number of the I/O driver for the drive identified by ioVDrvInfo.
ioVFSID	The file-system identifier. It indicates which file system is servicing the volume; it's zero for File Manager volumes and nonzero for volumes handled by an external file system.
ioVBkUp	The date and time the volume was last backed up (it's 0 if never backed up).
ioVSeqNum	Used internally.
ioVWrCnt	The volume write count.
ioVFilCnt	The total number of files on the volume.
ioVDirCnt	The total number of directories (not including the root directory) on the volume.
ioVFndrInfo	Information used by the Finder.

Field descriptions for the accessParam variant

filler3	Reserved.
ioDenyModes	Access mode information. The bits in this field have these meanings:

Bit	Meaning
0	If set, request read permission
1	If set, request write permission
2–3	Reserved; must be 0
4	If set, deny other readers access to this file
5	If set, deny other writers access to this file
6–15	Reserved; must be 0

filler4	Reserved.
filler5	Reserved.
ioACUser	The user's access rights for the specified directory. The bits in this field have the following meanings:

Bit	Meaning
0	Set if user does not have See Folder privileges
1	Set if user does not have See Files privileges
2	Set if user does not have Make Changes privileges
3–6	Reserved; always set to 0
7	Set if user is not owner of the directory

filler6	Reserved.
ioACOwnerID	The owner ID.
ioACGroupID	The group ID.
ioACAccess	The directory access privileges. See the section "Directory Access Privileges," beginning on page 2-18, for a complete description of this field.

Field descriptions for the `objParam` variant

`filler7` Reserved.

`ioObjType` A function code. The values passed in this field are determined by the routine to which you pass this parameter block.

`ioObjNamePtr` A pointer to the returned creator / group name.

`ioObjID` The creator / group ID.

Field descriptions for the `copyParam` variant

`ioDstVRefNum` A volume reference number for the destination volume.

`filler8` Reserved.

`ioNewName` A pointer to the destination pathname.

`ioCopyName` A pointer to an optional name.

`ioNewDirID` A destination directory ID.

Field descriptions for the `wdParam` variant

`filler9` Reserved.

`ioWDIndex` An index to working directories.

`ioWDProcID` The working directory user identifier.

`ioWDVRefNum` The volume reference number for the working directory.

`filler10` Reserved.

`filler11` Reserved.

`filler12` Reserved.

`filler13` Reserved.

`ioWDDirID` The working directory's directory ID.

Field descriptions for the `fidParam` variant

`filler14` Reserved.

`ioDestNamePtr` A pointer to the name of the destination file.

`filler15` Reserved.

`ioDestDirID` The parent directory ID of the destination file.

`filler16` Reserved.

`filler17` Reserved.

`ioSrcDirID` The parent directory ID of the source file.

`filler18` Reserved.

`ioFileID` The file ID.

Field descriptions for the `csParam` variant

`ioMatchPtr` A pointer to an array of `FSSpec` records in which the file and directory names that match the selection criteria are returned. The array must be large enough to hold the largest possible number of `FSSpec` records, as determined by the `ioReqMatchCount` field.

`ioReqMatchCount`
The maximum number of matches to return. This number should be the number of `FSSpec` records that will fit in the memory pointed

to by `ioMatchPtr`. You can use this field to avoid a possible excess of matches for criteria that prove to be too general (or to limit the length of a search if the `ioSearchTime` field isn't used).

`ioActMatchCount`
The number of actual matches found.

`ioSearchBits` The fields of the parameter blocks `ioSearchInfo1` and `ioSearchInfo2` that are relevant to the search. See "Searching a Volume" beginning on page 2-39 for constants you can add to determine a value for `ioSearchBits`.

`ioSearchInfo1` A pointer to a `CInfoPBRec` parameter block that contains values and the lower bounds of ranges for the fields selected by `ioSearchBits`.

`ioSearchInfo2` A pointer to a second `CInfoPBRec` parameter block that contains masks and upper bounds of ranges for the fields selected by `ioSearchBits`.

`ioSearchTime` A time limit on a search, in Time Manager format. Use this field to limit the run time of a single call to `PBCatSearch`. A value of 0 imposes no time limit. If the value of this field is positive, it is interpreted as milliseconds. If the value of this field is negative, it is interpreted as negated microseconds.

`ioCatPosition` A position in the catalog where searching should begin. Use this field to keep an index into the catalog when breaking down the `PBCatSearch` search into a number of smaller searches. This field is valid whenever `PBCatSearch` exits because it either spends the maximum time allowed by `ioSearchTime` or finds the maximum number of matches allowed by `ioReqMatchCount`.

To start at the beginning of the catalog, set the `initialize` field of `ioCatPosition` to 0. Before exiting after an interrupted search, `PBCatSearch` sets that field to the next catalog entry to be searched.

To resume where the previous call stopped, pass the entire `CatPosition` record returned by the previous call as input to the next.

`ioOptBuffer` A pointer to an optional read buffer. The `ioOptBuffer` and `ioOptBufSize` fields let you specify a part of memory as a read buffer, increasing search speed.

`ioOptBufSize` The size of the buffer pointed to by `ioOptBuffer`. Buffer size effectiveness varies with models and configurations, but a 16 KB buffer is likely to be optimal. The size should be at least 1024 bytes and should be an integral multiple of 512 bytes.

Field descriptions for the `foreignPrivParam` variant

`filler21` Reserved.

`filler22` Reserved.

`ioForeignPrivBuffer`
A pointer to a buffer containing access-control information about the foreign file system.

ioForeignPrivReqCount
: The size of the buffer pointed to by the ioForeignPrivBuffer field.

ioForeignPrivActCount
: The amount of the buffer pointed to by the ioForeignPrivBuffer field that was actually used to hold data.

filler23
: Reserved.

ioForeignPrivDirID
: The parent directory ID of the foreign file or directory.

ioForeignPrivInfo1
: A long word that may contain privileges data.

ioForeignPrivInfo2
: A long word that may contain privileges data.

ioForeignPrivInfo3
: A long word that may contain privileges data.

ioForeignPrivInfo4
: A long word that may contain privileges data.

Catalog Information Parameter Blocks

The low-level functions PBGetCatInfo, PBSetCatInfo, and PBCatSearch exchange information with your application using the catalog information parameter block, which is defined by the CInfoPBRec data type. There are two variants of this record, hFileInfo and dirInfo, which describe files and directories, respectively.

```
TYPE CInfoPBRec   =  {catalog information parameter block}
RECORD
   qLink:            QElemPtr;      {next queue entry}
   qType:            Integer;       {queue type}
   ioTrap:           Integer;       {routine trap}
   ioCmdAddr:        Ptr;           {routine address}
   ioCompletion:     ProcPtr;       {pointer to completion routine}
   ioResult:         OSErr;         {result code}
   ioNamePtr:        StringPtr;     {pointer to pathname}
   ioVRefNum:        Integer;       {volume specification}
   ioFRefNum:        Integer;       {file reference number}
   ioFVersNum:       SignedByte;    {version number}
   filler1:          SignedByte;    {reserved}
   ioFDirIndex:      Integer;       {directory index}
   ioFlAttrib:       SignedByte;    {file or directory attributes}
   ioACUser:         SignedByte;    {directory access rights}
CASE CInfoType OF
hFileInfo:
  (ioFlFndrInfo:     FInfo;         {information used by the Finder}
   ioDirID:          LongInt;       {directory ID or file ID}
   ioFlStBlk:        Integer;       {first alloc. blk. of data fork}
```

```
    ioFlLgLen:        LongInt;       {logical EOF of data fork}
    ioFlPyLen:        LongInt;       {physical EOF of data fork}
    ioFlRStBlk:       Integer;       {first alloc. blk. of resource fork}
    ioFlRLgLen:       LongInt;       {logical EOF of resource fork}
    ioFlRPyLen:       LongInt;       {physical EOF of resource fork}
    ioFlCrDat:        LongInt;       {date and time of creation}
    ioFlMdDat:        LongInt;       {date and time of last modification}
    ioFlBkDat:        LongInt;       {date and time of last backup}
    ioFlXFndrInfo:    FXInfo;        {additional Finder information}
    ioFlParID:        LongInt;       {file parent directory ID}
    ioFlClpSiz:       LongInt);      {file's clump size}
dirInfo:
  (ioDrUsrWds:        DInfo;         {information used by the Finder}
    ioDrDirID:        LongInt;       {directory ID}
    ioDrNmFls:        Integer;       {number of files in directory}
    filler3:          ARRAY[1..9] OF Integer;
    ioDrCrDat:        LongInt;       {date and time of creation}
    ioDrMdDat:        LongInt;       {date and time of last modification}
    ioDrBkDat:        LongInt;       {date and time of last backup}
    ioDrFndrInfo:     DXInfo;        {additional Finder information}
    ioDrParID:        LongInt);      {directory's parent directory ID}
END;
```

The first 14 fields are common to both variants. Each variant also includes its own unique fields.

Field descriptions common to both variants

`qLink` A pointer to the next entry in the file I/O queue. (This field is used internally by the File Manager to keep track of asynchronous calls awaiting execution.)

`qType` The queue type. (This field is used internally by the File Manager.)

`ioTrap` The trap number of the routine that was called. (This field is used internally by the File Manager.)

`ioCmdAddr` The address of the routine that was called. (This field is used internally by the File Manager.)

`ioCompletion` A pointer to a completion routine to be executed at the end of an asynchronous call. It should be `NIL` for asynchronous calls with no completion routine and is automatically set to `NIL` for all synchronous calls. See “Completion Routines” on page 2-240 for information about completion routines.

`ioResult` The result code of the function. For synchronous calls, this field is the same as the result code of the function call itself. To determine when an asynchronous call has actually been completed, your application can poll this field; it’s set to a positive number when the call is made and receives the actual result code when the call is completed.

`ioNamePtr` A pointer to a pathname. Whenever a routine description specifies that `ioNamePtr` is used—whether for input, output, or both—it's very important that you set this field to point to storage for a `Str255` value (if you're using a pathname) or to `NIL` (if you're not).

`ioVRefNum` A volume specification. You can specify a volume using a volume reference number, a drive number, a working directory reference number, or 0 for the default drive.

`ioFRefNum` The file reference number of an open file.

`ioFVersNum` A file version number. This field is no longer used and you should always set it to 0.

`filler1` Reserved.

`ioFDirIndex` A file and directory index. If this field contains a positive number, `PBGetCatInfo` returns information about the file or directory having that directory index in the directory specified by the `ioVRefNum` field. (If `ioVRefNum` contains a volume reference number, the specified directory is that volume's root directory.)

If this field contains 0, `PBGetCatInfo` returns information about the file or directory whose name is specified in the `ioNamePtr` field and that is located in the directory specified by the `ioVRefNum` field. (Once again, if `ioVRefNum` contains a volume reference number, the specified directory is that volume's root directory.)

If this field contains a negative number, `PBGetCatInfo` ignores the `ioNamePtr` field and returns information about the directory specified in the `ioDirID` field. If both `ioDirID` and `ioVRefNum` are set to 0, `PBGetCatInfo` returns information about the current default directory.

`ioFlAttrib` File or directory attributes. For files, the bits in this field have the following meanings:

Bit	Meaning
0	Set if file is locked
1	Reserved
2	Set if resource fork is open
3	Set if data fork is open
4	Set if a directory
5–6	Reserved
7	Set if file (either fork) is open

For directories, the bits in this field have the following meanings:

Bit	Meaning
0	Set if the directory is locked
1	Reserved
2	Set if the directory is within a shared area of the directory hierarchy
3	Set if the directory is a share point that is mounted by some user

Bit	Meaning
4	Set if the item is a directory
5	Set if the directory is a share point
6–7	Reserved

`ioACUser` The user's access rights for the specified directory. The bits in this field have the following meanings:

Bit	Meaning
0	Set if user does not have See Folder privileges
1	Set if user does not have See Files privileges
2	Set if user does not have Make Changes privileges
3–6	Reserved; always set to 0
7	Set if user is not owner of the directory

For example, if you call `PBGetCatInfo` for a particular shared volume and `ioACUser` returns 0, you know that the user is the owner of the directory and has complete privileges to it.

Field descriptions for the `hFileInfo` variant

`ioFlFndrInfo` Information used by the Finder.

`ioDirID` A directory ID or file ID. On input to `PBGetCatInfo`, this field contains a directory ID (which is used only if the `ioFDirIndex` field is negative). On output, this field contains the file ID of the specified file.

`ioFlStBlk` The first allocation block of the data fork. This field contains 0 if the file's data fork is empty.

`ioFlLgLen` The logical end-of-file of the data fork.

`ioFlPyLen` The physical end-of-file of the data fork.

`ioFlRStBlk` The first allocation block of the resource fork.

`ioFlRLgLen` The logical end-of-file of the resource fork.

`ioFlRPyLen` The physical end-of-file of the resource fork.

`ioFlCrDat` The date and time of the file's creation, specified in seconds since midnight, January 1, 1904.

`ioFlMdDat` The date and time of the last modification to the file, specified in seconds since midnight, January 1, 1904.

`ioFlBkDat` The date and time of the last backup to the file, specified in seconds since midnight, January 1, 1904.

`ioFlXFndrInfo` Additional information used by the Finder. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for details.)

`ioFlParID` The directory ID of the file's parent.

`ioFlClpSiz` The clump size to be used when writing the file; if it's 0, the volume's clump size is used when the file is opened.

Field descriptions for the `dirInfo` variant

`ioDrUsrWds`	Information used by the Finder.
`ioDrDirID`	A directory ID. On input to `PBGetCatInfo`, this field contains a directory ID (which is used only if the value of the `ioFDirIndex` field is negative). On output, this field contains the directory ID of the specified directory.
`ioDrNmFls`	The number of files in the directory.
`filler3`	Reserved.
`ioDrCrDat`	The date and time of the directory's creation, specified in seconds since midnight, January 1, 1904.
`ioDrMdDat`	The date and time of the last modification to the directory, specified in seconds since midnight, January 1, 1904.
`ioDrBkDat`	The date and time of the last backup to the directory, specified in seconds since midnight, January 1, 1904.
`ioDrFndrInfo`	Additional information used by the Finder.
`ioDrParID`	The directory ID of the specified directory's parent.

Catalog Position Records

When you call the `PBCatSearch` function to search a volume's catalog file, you can specify (in the `ioCatPosition` field of the parameter block passed to `PBCatSearch`) a catalog position record. If a catalog search consumes more time than is allowed by the `ioSearchTime` field, `PBCatSearch` stores a directory-location index in that record; when you call `PBCatSearch` again, it uses that record to resume searching where it left off. A catalog position record is defined by the `CatPositionRec` data type.

```
TYPE CatPositionRec  =  {catalog position record}
RECORD
   initialize:    LongInt;                {starting point}
   priv:          ARRAY[1..6] OF Integer; {private data}
END;
```

Field descriptions

`initialize`	The starting point of the catalog search. To start searching at the beginning of a catalog, specify 0 in this field. To resume a previous search, pass the value returned by the previous call to `PBCatSearch`.
`priv`	An array of integers that is used internally by `PBCatSearch`.

Catalog Move Parameter Blocks

The low-level HFS function `PBCatMove` uses the catalog move parameter block defined by the `CMovePBRec` data type.

```
TYPE CMovePBRec   =           {catalog move parameter block}
RECORD
   qLink:         QElemPtr;   {next queue entry}
   qType:         Integer;    {queue type}
   ioTrap:        Integer;    {routine trap}
   ioCmdAddr:     Ptr;        {routine address}
   ioCompletion:  ProcPtr;    {pointer to completion routine}
   ioResult:      OSErr;      {result code}
   ioNamePtr:     StringPtr;  {pointer to pathname}
   ioVRefNum:     Integer;    {volume specification}
   filler1:       LongInt;    {reserved}
   ioNewName:     StringPtr;  {name of new directory}
   filler2:       LongInt;    {reserved}
   ioNewDirID:    LongInt;    {directory ID of new directory}
   filler3:       ARRAY[1..2] OF LongInt; {reserved}
   ioDirID:       LongInt;    {directory ID of current directory}
END;
```

Field descriptions

`qLink` A pointer to the next entry in the file I/O queue. (This field is used internally by the File Manager to keep track of asynchronous calls awaiting execution.)

`qType` The queue type. (This field is used internally by the File Manager.)

`ioTrap` The trap number of the routine that was called. (This field is used internally by the File Manager.)

`ioCmdAddr` The address of the routine that was called. (This field is used internally by the File Manager.)

`ioCompletion` A pointer to a completion routine to be executed at the end of an asynchronous call. It should be `NIL` for asynchronous calls with no completion routine and is automatically set to `NIL` for all synchronous calls. See “Completion Routines” on page 2-240 for information about completion routines.

`ioResult` The result code of the function. For synchronous calls, this field is the same as the result code of the function call itself. To determine when an asynchronous call has actually been completed, your application can poll this field; it’s set to a positive number when the call is made and receives the actual result code when the call is completed.

`ioNamePtr` A pointer to a pathname. Whenever a routine description specifies that `ioNamePtr` is used—whether for input, output, or both—it’s very important that you set this field to point to storage for a `Str255` value (if you’re using a pathname) or to `NIL` (if you’re not).

`ioVRefNum` A volume specification (volume reference number, working directory reference number, drive number, or 0 for default volume).

`filler1` Reserved.

ioNewName	The name of the directory into which the specified file or directory is to be moved.
filler2	Reserved.
ioNewDirID	The directory ID of the directory into which the specified file or directory is to be moved.
filler3	Reserved.
ioDirID	The current directory ID of the file or directory to be moved (used in conjunction with the ioVRefNum and ioNamePtr fields).

Working Directory Parameter Blocks

The low-level HFS functions PBOpenWD, PBCloseWD, and PBGetWDInfo use the working directory parameter block defined by the WDPBRec data type.

```
TYPE WDPBRec      =           {working directory parameter block}
RECORD
   qLink:         QElemPtr;   {next queue entry}
   qType:         Integer;    {queue type}
   ioTrap:        Integer;    {routine trap}
   ioCmdAddr:     Ptr;        {routine address}
   ioCompletion:  ProcPtr;    {pointer to completion routine}
   ioResult:      OSErr;      {result code}
   ioNamePtr:     StringPtr;  {pointer to pathname}
   ioVRefNum:     Integer;    {volume specification}
   filler1:       Integer;    {reserved}
   ioWDIndex:     Integer;    {working directory index}
   ioWDProcID:    LongInt;    {working directory user identifier}
   ioWDVRefNum:   Integer;    {working directory's vol. ref. num.}
   filler2:       ARRAY[1..7] OF Integer; {reserved}
   ioWDDirID:     LongInt;    {working directory's directory ID}
END;
```

Field descriptions

qLink	A pointer to the next entry in the file I/O queue. (This field is used internally by the File Manager to keep track of asynchronous calls awaiting execution.)
qType	The queue type. (This field is used internally by the File Manager.)
ioTrap	The trap number of the routine that was called. (This field is used internally by the File Manager.)
ioCmdAddr	The address of the routine that was called. (This field is used internally by the File Manager.)
ioCompletion	A pointer to a completion routine to be executed at the end of an asynchronous call. It should be NIL for asynchronous calls with no completion routine and is automatically set to NIL for all synchronous calls. See “Completion Routines” on page 2-240 for information about completion routines.

ioResult	The result code of the function. For synchronous calls, this field is the same as the result code of the function call itself. To determine when an asynchronous call has actually been completed, your application can poll this field; it's set to a positive number when the call is made and receives the actual result code when the call is completed.
ioNamePtr	A pointer to a pathname. Whenever a routine description specifies that ioNamePtr is used—whether for input, output, or both—it's very important that you set this field to point to storage for a Str255 value (if you're using a pathname) or to NIL (if you're not).
ioVRefNum	A volume specification (volume reference number, working directory reference number, drive number, or 0 for default volume).
filler1	Reserved.
ioWDIndex	An index for use with the PBGetWDInfo function.
ioWDProcID	An identifier that's used to distinguish between working directories set up by different users; you should set ioWDProcID to your application's signature.
ioWDVRefNum	The working directory's volume reference number.
filler2	Reserved.
ioWDDirID	The working directory's directory ID.

File Control Block Parameter Blocks

The low-level HFS function PBGetFCBInfo uses the file control block parameter block defined by the FCBPBRec data type.

```
TYPE FCBPBRec     =        {file control block parameter block}
RECORD
   qLink:         QElemPtr;      {next queue entry}
   qType:         Integer;       {queue type}
   ioTrap:        Integer;       {routine trap}
   ioCmdAddr:     Ptr;           {routine address}
   ioCompletion:  ProcPtr;       {pointer to completion routine}
   ioResult:      OSErr;         {result code}
   ioNamePtr:     StringPtr;     {pointer to pathname}
   ioVRefNum:     Integer;       {volume specification}
   ioRefNum:      Integer;       {file reference number}
   filler:        Integer;       {reserved}
   ioFCBIndx:     Integer;       {FCB index}
   filler1:       Integer;       {reserved}
   ioFCBFlNm:     LongInt;       {file ID}
   ioFCBFlags:    Integer;       {flags}
   ioFCBStBlk:    Integer;       {first allocation block of file}
   ioFCBEOF:      LongInt;       {logical end-of-file}
   ioFCBPLen:     LongInt;       {physical end-of-file}
```

```
   ioFCBCrPs:     LongInt;       {position of the file mark}
   ioFCBVRefNum:  Integer;       {volume reference number}
   ioFCBClpSiz:   LongInt;       {file's clump size}
   ioFCBParID:    LongInt;       {parent directory ID}
END;
```

Field descriptions

`qLink` A pointer to the next entry in the file I/O queue. (This field is used internally by the File Manager to keep track of asynchronous calls awaiting execution.)

`qType` The queue type. (This field is used internally by the File Manager.)

`ioTrap` The trap number of the routine that was called. (This field is used internally by the File Manager.)

`ioCmdAddr` The address of the routine that was called. (This field is used internally by the File Manager.)

`ioCompletion` A pointer to a completion routine to be executed at the end of an asynchronous call. It should be `NIL` for asynchronous calls with no completion routine and is automatically set to `NIL` for all synchronous calls. See "Completion Routines" on page 2-240 for information about completion routines.

`ioResult` The result code of the function. For synchronous calls, this field is the same as the result code of the function call itself. To determine when an asynchronous call has actually been completed, your application can poll this field; it's set to a positive number when the call is made and receives the actual result code when the call is completed.

`ioNamePtr` A pointer to a pathname. Whenever a routine description specifies that `ioNamePtr` is used—whether for input, output, or both—it's very important that you set this field to point to storage for a `Str255` value (if you're using a pathname) or to `NIL` (if you're not).

`ioVRefNum` A volume specification (volume reference number, working directory reference number, drive number, or 0 for default volume).

`ioRefNum` The file reference number of an open file.

`filler` Reserved.

`ioFCBIndx` An index for use with the `PBGetFCBInfo` function.

`filler1` Reserved.

`ioFCBFlNm` The file ID.

`ioFCBFlags` Flags describing the status of the file. The bits in this field that are currently used have the following meanings:

Bit	Meaning
8	Set if data can be written to the file
9	Set if this FCB describes a resource fork
10	Set if the file has a locked byte range
11	Reserved

Bit	Meaning
12	Set if the file has shared write permissions
13	Set if the file is locked (write-protected)
14	Set if the file's clump size is specified in the FCB
15	Set if the file has changed since it was last flushed

`ioFCBStBlk` The number of the first allocation block of the file.

`ioFCBEOF` The logical end-of-file.

`ioFCBPLen` The physical end-of-file.

`ioFCBCrPs` The position of the file mark.

`ioFCBVRefNum` The volume reference number.

`ioFCBClpSiz` The file clump size.

`ioFCBParID` The file's parent directory ID.

Volume Attributes Buffer

The low-level HFS function `PBHGetVolParms` returns information in the volume attributes buffer, defined by the `GetVolParmsInfoBuffer` data type.

```
TYPE GetVolParmsInfoBuffer =
RECORD
   vMVersion:        Integer;    {version number}
   vMAttrib:         LongInt;    {volume attributes}
   vMLocalHand:      Handle;     {reserved}
   vMServerAdr:      LongInt;    {network server address}
   vMVolumeGrade:    LongInt;    {relative speed rating}
   vMForeignPrivID:  Integer;    {foreign privilege model}
END;
```

Field descriptions

`vMVersion` The version of the attributes buffer structure. Currently this field returns either 1 or 2.

`vMAttrib` A 32-bit quantity that encodes information about the volume attributes. See the list of constants in the description of `PBHGetVolParms` beginning on page 2-148 for details on the meaning of each bit.

`vMLocalHand` A handle to private data for shared volumes. On creation of the VCB (right after mounting), this field is a handle to a 2-byte block of memory. The Finder uses this for its local window list storage, allocating and deallocating memory as needed. It is disposed of when the volume is unmounted. Your application should treat this field as reserved.

`vMServerAdr` For AppleTalk server volumes, this field contains the internet address of an AppleTalk server volume. Your application can

inspect this field to tell which volumes belong to which server; the value of this field is 0 if the volume does not have a server.

`vMVolumeGrade` The relative speed rating of the volume. The scale used to determine these values is currently uncalibrated. In general, lower values indicate faster speeds. A value of 0 indicates that the volume's speed is unrated. The buffer version returned in the `vMVersion` field must be greater than 1 for this field to be meaningful.

`vMForeignPrivID`
An integer representing the privilege model supported by the volume. Currently two values are defined for this field: 0 represents a standard HFS volume that might or might not support the AFP privilege model; `fsUnixPriv` represents a volume that supports the A/UX privilege model. The buffer version returned in the `vMVersion` field must be greater than 1 for this field to be meaningful.

Volume Mounting Information Records

The File Manager remote mounting functions store the mounting information in a variable-sized structure called a volume mounting information record, defined by the `VolMountInfoHeader` data type.

```
TYPE VolMountInfoHeader =        {volume mounting information}
RECORD
   length:           Integer;    {length of mounting information}
   media:            VolumeType; {type of volume}
   {volume-specific, variable-length location data}
END;
```

Field descriptions

`length` The length of the `VolMountInfoHeader` structure (that is, the total length of the structure header described here plus the variable-length location data). The length of the record is flexible so that non-Macintosh file systems can store whatever information they need for volume mounting.

`media` The volume type of the remote volume. The value `AppleShareMediaType` (a constant that translates to `'afpm'`) represents an AppleShare volume. If you are adding support for the programmatic mounting functions to a non-Macintosh file system, you should register a four-character identifier for your volumes with Macintosh Developer Technical Support at Apple Computer, Inc.

The only volumes that currently support the programmatic mounting functions are AppleShare servers, which use a volume mounting record of type `AFPVolMountInfo`.

```
TYPE AFPVolMountInfo    =       {AFP volume mounting information}
RECORD
   length:           Integer;    {length of mounting information}
   media:            VolumeType; {type of volume}
   flags:            Integer;    {reserved; must be set to 0}
   nbpInterval:      SignedByte; {NBP retry interval}
   nbpCount:         SignedByte; {NBP retry count}
   uamType:          Integer;    {user authentication method}
   zoneNameOffset:   Integer;    {offset to zone name}
   serverNameOffset: Integer;    {offset server name}
   volNameOffset:    Integer;    {offset to volume name}
   userNameOffset:   Integer;    {offset to user name}
   userPasswordOffset:
                     Integer;    {offset to user password}
   volPasswordOffset:
                     Integer;    {offset to volume password}
   AFPData:          PACKED ARRAY[1..144] OF CHAR;
                                 {standard AFP mounting info}
   {optional volume-specific, variable-length data}
END;
```

Field descriptions

`length` The length of the `AFPVolMountInfo` structure (that is, the total length of the structure header described here plus the variable-length location data).

`media` The volume type of the remote volume. The value `AppleShareMediaType` (a constant that translates to `'afpm'`) represents an AppleShare volume.

`flags` Reserved; set this field to 0. If bit 0 is set, no greeting message from the server is displayed.

`nbpInterval` The NBP retransmit interval, in units of 8 ticks.

`npbCount` The NBP retransmit count. This field specifies the *total* number of times a packet should be transmitted, including the first transmission.

`uamType` The access-control method used by the remote volume. AppleShare uses four methods, defined by constants:

```
CONST
   kNoUserAuthentication   = 1;  {no password}
   kPassword               = 2;  {8-byte password}
   kEncryptPassword        = 3;
                          {encrypted 8-byte password}
   kTwoWayEncryptPassword  = 6;
                          {two-way random encryption}
```

`zoneNameOffset` The offset in bytes from the beginning of the record to the entry in the `AFPData` field containing the name of the AppleShare zone.

`serverNameOffset` The offset in bytes from the beginning of the record to the entry in the `AFPData` field containing the name of the AppleShare server.

`volNameOffset` The offset in bytes from the beginning of the record to the entry in the `AFPData` field containing the name of the volume.

`userNameOffset` The offset in bytes from the beginning of the record to the entry in the `AFPData` field containing the name of the user.

`userPasswordOffset` The offset in bytes from the beginning of the record to the entry in the `AFPData` field containing the user's password.

`volPasswordOffset` The offset in bytes from the beginning of the record to the entry in the `AFPData` field containing the volume's password. Some versions of the AppleShare software do not pass the information in this field to the server.

`AFPData` The actual volume mounting information, offsets to which are contained in the preceding six fields. To mount an AFP volume, you must fill in the record with at least the zone name, server name, user name, user password, and volume password. You can lay out the data in any order within this data field, as long as you specify the correct offsets in the offset fields.

High-Level File Access Routines

This section describes the File Manager's high-level file access routines. When you call one of these routines, you specify a file by a file reference number (which the File Manager returns to your application when the application opens a file). Unless your application has very specialized needs, you should be able to manage all file access (for example, writing data to the file) using the routines described in this section. Typically you use these routines to operate on a file's data fork, but in certain circumstances you might want to use them on a file's resource fork as well.

Reading, Writing, and Closing Files

You can use the functions `FSRead`, `FSWrite`, and `FSClose` to read data from a file, write data to a file, and close an open file. All three of these functions operate on open files. You can use any one of a variety of routines to open a file (for example, `FSpOpenDF`).

FSRead

You can use the FSRead function to read any number of bytes from an open file.

```
FUNCTION FSRead (refNum: Integer; VAR count: LongInt;
                 buffPtr: Ptr): OSErr;
```

refNum	The file reference number of an open file.
count	On input, the number of bytes to read; on output, the number of bytes actually read.
buffPtr	A pointer to the data buffer into which the bytes are to be read.

DESCRIPTION

The FSRead function attempts to read the requested number of bytes from the specified file into the specified buffer. The buffPtr parameter points to that buffer; this buffer is allocated by your application and must be at least as large as the count parameter.

Because the read operation begins at the current mark, you might want to set the mark first by calling the SetFPos function. If you try to read past the logical end-of-file, FSRead reads in all the data up to the end-of-file, moves the mark to the end-of-file, and returns eofErr as its function result. Otherwise, FSRead moves the file mark to the byte following the last byte read and returns noErr.

Note

The low-level PBRead function lets you set the mark without having to call SetFPos. Also, if you want to read data in newline mode, you must use PBRead instead of FSRead. ◆

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
fnOpnErr	–38	File not open
eofErr	–39	Logical end-of-file reached
posErr	–40	Attempt to position mark before start of file
fLckdErr	-45	File is locked
paramErr	–50	Negative count
rfNumErr	–51	Bad reference number
afpAccessDenied	–5000	User does not have the correct access to the file

FSWrite

You can use the `FSWrite` function to write any number of bytes to an open file.

```
FUNCTION FSWrite (refNum: Integer; VAR count: LongInt;
                  buffPtr: Ptr): OSErr;
```

`refNum`	The file reference number of an open file.
`count`	On input, the number of bytes to write to the file; on output, the number of bytes actually written.
`buffPtr`	A pointer to the data buffer from which the bytes are to be written.

DESCRIPTION

The `FSWrite` function takes the specified number of bytes from the specified data buffer and attempts to write them to the specified file. Because the write operation begins at the current mark, you might want to set the mark first by calling the `SetFPos` function.

If the write operation completes successfully, `FSWrite` moves the file mark to the byte following the last byte written and returns `noErr`. If you try to write past the logical end-of-file, `FSWrite` moves the logical end-of-file. If you try to write past the physical end-of-file, `FSWrite` adds one or more clumps to the file and moves the physical end-of-file accordingly.

Note
The low-level `PBWrite` function lets you set the mark without having to call `SetFPos`. ◆

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Disk full
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`posErr`	–40	Attempt to position mark before start of file
`wPrErr`	–44	Hardware volume lock
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Software volume lock
`paramErr`	–50	Negative `count`
`rfNumErr`	–51	Bad reference number
`wrPermErr`	–61	Read/write permission doesn't allow writing

FSClose

You can use the `FSClose` function to close an open file.

```
FUNCTION FSClose (refNum: Integer): OSErr;
```

`refNum` The file reference number of an open file.

DESCRIPTION

The `FSClose` function removes the access path for the specified file and writes the contents of the volume buffer to the volume.

Note

The `FSClose` function calls `PBFlushFile` internally to write the file's bytes onto the volume. To ensure that the file's catalog entry is updated, you should call `FlushVol` after you call `FSClose`. ◆

▲ **WARNING**

Make sure that you do not call `FSClose` with a file reference number of a file that has already been closed. Attempting to close the same file twice may result in loss of data on a volume. See "File Control Blocks" on page 2-82 for a description of how this can happen. ▲

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`fnfErr`	–43	File not found
`rfNumErr`	–51	Bad reference number

Manipulating the File Mark

You can use the functions `GetFPos` and `SetFPos` to get or set the current position of the file mark.

GetFPos

You can use the `GetFPos` function to determine the current position of the mark before reading from or writing to an open file.

```
FUNCTION GetFPos (refNum: Integer; VAR filePos: LongInt): OSErr;
```

`refNum` The file reference number of an open file.

`filePos` On output, the current position of the mark.

DESCRIPTION

The `GetFPos` function returns, in the `filePos` parameter, the current position of the file mark for the specified open file. The position value is zero-based; that is, the value of `filePos` is 0 if the file mark is positioned at the beginning of the file.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
fnOpnErr	–38	File not open
rfNumErr	–51	Bad reference number
gfpErr	–52	Error during GetFPos

SetFPos

You can use the SetFPos function to set the position of the file mark before reading from or writing to an open file.

```
FUNCTION SetFPos (refNum: Integer; posMode: Integer;
                  posOff: LongInt): OSErr;
```

refNum	The file reference number of an open file.
posMode	The positioning mode.
posOff	The positioning offset.

DESCRIPTION

The SetFPos function sets the file mark of the specified file. The posMode parameter indicates how to position the mark; it must contain one of the following values:

```
CONST
   fsAtMark    = 0;  {at current mark}
   fsFromStart = 1;  {set mark relative to beginning of file}
   fsFromLEOF  = 2;  {set mark relative to logical end-of-file}
   fsFromMark  = 3;  {set mark relative to current mark}
```

If you specify fsAtMark, the mark is left wherever it's currently positioned, and the posOff parameter is ignored. The next three constants let you position the mark relative to either the beginning of the file, the logical end-of-file, or the current mark. If you specify one of these three constants, you must also pass in posOff a byte offset (either positive or negative) from the specified point. If you specify fsFromLEOF, the value in posOff must be less than or equal to 0.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
fnOpnErr	–38	File not open
eofErr	–39	Logical end-of-file reached
posErr	–40	Attempt to position mark before start of file
rfNumErr	–51	Bad reference number

Manipulating the End-of-File

You can use the functions `GetEOF` and `SetEOF` to get or set the logical end-of-file of an open file.

GetEOF

You can use the `GetEOF` function to determine the current logical end-of-file of an open file.

```
FUNCTION GetEOF (refNum: Integer; VAR logEOF: LongInt): OSErr;
```

`refNum` The file reference number of an open file.

`logEOF` On output, the logical end-of-file.

DESCRIPTION

The `GetEOF` function returns, in the `logEOF` parameter, the logical end-of-file of the specified file.

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`rfNumErr`	–51	Bad reference number
`afpAccessDenied`	–5000	User does not have the correct access to the file

SetEOF

You can use the `SetEOF` function to set the logical end-of-file of an open file.

```
FUNCTION SetEOF (refNum: Integer; logEOF: LongInt): OSErr;
```

`refNum` The file reference number of an open file.

`logEOF` The logical end-of-file.

DESCRIPTION

The `SetEOF` function sets the logical end-of-file of the specified file. If you attempt to set the logical end-of-file beyond the physical end-of-file, the physical end-of-file is set 1 byte beyond the end of the next free allocation block; if there isn't enough space on the volume, no change is made, and `SetEOF` returns `dskFulErr` as its function result.

If you set the `logEOF` parameter to 0, all space occupied by the file on the volume is released. The file still exists, but it contains 0 bytes. Setting a file fork's end-of-file to 0 is therefore not the same as deleting the file (which removes both file forks at once).

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Disk full
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`wPrErr`	–44	Hardware volume lock
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Software volume lock
`rfNumErr`	–51	Bad reference number
`wrPermErr`	–61	Read/write permission doesn't allow writing

Allocating File Blocks

The File Manager provides two functions, `Allocate` and `AllocContig`, that allow you to allocate additional blocks to a file. The File Manager automatically allocates file blocks if you move the logical end-of-file past the physical end-of-file, and it automatically deallocates unneeded blocks from a file if you move the logical end-of-file to a position more than one allocation block before the current physical end-of-file. Consequently, you do not in general need to be concerned with allocating or deallocating file blocks. However, you can improve file block contiguity if you use the `Allocate` or `AllocContig` function to preallocate file blocks. This is most useful if you know in advance how big a file is likely to become.

Note

When the File Manager allocates (or deallocates) file blocks automatically, it always adds (or removes) blocks in clumps. The `Allocate` and `AllocContig` functions allow you to add blocks in allocation blocks, which may be smaller than clumps. ◆

The `Allocate` and `AllocContig` functions are not supported by AppleShare volumes. Instead, use `SetEOF` or `PBSetEOF` to extend a file by setting the end-of-file.

Allocate

You can use the `Allocate` function to allocate additional blocks to an open file.

```
FUNCTION Allocate (refNum: Integer; VAR count: LongInt): OSErr;
```

`refNum`	The file reference number of an open file.
`count`	On input, the number of additional bytes to allocate to the file; on output, the number of bytes actually allocated, rounded up to the nearest multiple of the allocation block size.

DESCRIPTION

The `Allocate` function adds the specified number of bytes to the specified file and sets the physical end-of-file to 1 byte beyond the last block allocated. If there isn't enough empty space on the volume to satisfy the allocation request, `Allocate` allocates the rest of the space on the volume and returns `dskFulErr` as its function result.

The `Allocate` function always attempts to allocate contiguous blocks. If the total number of requested bytes is unavailable, `Allocate` allocates whatever space, contiguous or not, is available. To force the allocation of the entire requested space as a contiguous piece, call `AllocContig` instead.

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Disk full
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`wPrErr`	–44	Hardware volume lock
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Software volume lock
`rfNumErr`	–51	Bad reference number
`wrPermErr`	–61	Read/write permission doesn't allow writing

AllocContig

You can use the `AllocContig` function to allocate additional contiguous blocks to an open file.

```
FUNCTION AllocContig (refNum: Integer; VAR count: LongInt): OSErr;
```

`refNum` The file reference number of an open file.

`count` On input, the number of additional bytes to allocate to the file; on output, the number of bytes allocated, rounded up to the nearest multiple of the allocation block size.

DESCRIPTION

The `AllocContig` function is identical to the `Allocate` function except that if there isn't enough contiguous empty space on the volume to satisfy the allocation request, `AllocContig` does nothing and returns `dskFulErr` as its function result. If you want to allocate whatever space is available, even when the entire request cannot be filled by the allocation of a contiguous piece, call `Allocate` instead.

RESULT CODES

noErr	0	No error
dskFulErr	–34	Disk full
ioErr	–36	I/O error
fnOpnErr	–38	File not open
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
rfNumErr	–51	Bad reference number
wrPermErr	–61	Read/write permission doesn't allow writing

Low-Level File Access Routines

This section describes the low-level file access routines. These low-level routines, whose names begin with the letters PB, provide two advantages over the corresponding high-level file access routines:

- These routines can be executed asynchronously, returning control to your application before the operation is completed.
- In certain cases, these routines provide more extensive information or perform advanced operations.

All of these routines exchange parameters with your application through a parameter block of type ParamBlock. When you call a low-level routine, you pass the address of the parameter block to the routine.

Assembly-Language Note

When you call any of these low-level routines, register A0 must point to a parameter block containing the parameters for the routine. If you want the routine to be executed asynchronously, set bit 10 of the routine trap word. You can do this by supplying the word ASYNC as the second argument to the routine macro. Here's an example:

```
_Read, ASYNC
```

You can set or test bit 10 of a trap word using the global constant asyncTrpBit.

The hierarchical extensions of certain basic File Manager routines actually are not new calls. For instance, _Open and _HOpen both trap to the same routine. The trap word generated by the _HOpen macro is the same as the trap word that would be generated by invoking the _Open macro with bit 9 set. The setting of this bit tells the File Manager to expect a larger parameter block containing the additional fields (such as a directory ID) needed to handle a hierarchical directory volume. You can set or test bit 9 of a trap word by using the global constant hfsBit.

All File Manager routines return a result code in register D0. ◆

These low-level file access routines can run either synchronously or asynchronously. There are three versions of each routine. The first takes two parameters: a pointer to the

parameter block and a Boolean parameter that specifies whether the routine is to run asynchronously (TRUE) or synchronously (FALSE). For example, the first version of the low-level routine to read bytes from a file has this declaration:

```
FUNCTION PBRead (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
```

The second version does not take a second parameter; instead, it adds the suffix Async to the name of the routine.

```
FUNCTION PBReadAsync (paramBlock: ParmBlkPtr): OSErr;
```

Similarly, the third version of the routine does not take a second parameter; instead, it adds the suffix Sync to the name of the routine.

```
FUNCTION PBReadSync (paramBlock: ParmBlkPtr): OSErr;
```

Only the first version of each routine is documented in this section. (See "Summary of the File Manager," beginning on page 2-243, for a listing of all three versions of these routines.) Note, however, that the second and third versions of these routines do not use the glue code that the first version uses and are therefore more efficient.

Note

Although you can execute low-level file access routines asynchronously, the underlying device driver may not support asynchronous operation. The SCSI Manager, for example, currently supports only synchronous data transfers. Data transfers to a floppy disk or to a network server, however, can be made asynchronously. ◆

Reading, Writing, and Closing Files

You can use the functions PBRead, PBWrite, and PBClose to read data from a file, write data to a file, and close an open file. All three of these functions operate on open files. You can use any one of a variety of routines (for example, PBHOpenDF) to open a file.

PBRead

You can use the PBRead function to read any number of bytes from an open file.

```
FUNCTION PBRead (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
```

paramBlock A pointer to a basic File Manager parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioRefNum	Integer	A file reference number.
→	ioBuffer	Ptr	A pointer to a data buffer.
→	ioReqCount	LongInt	The number of bytes requested.
←	ioActCount	LongInt	The number of bytes actually read.
→	ioPosMode	Integer	The positioning mode.
↔	ioPosOffset	LongInt	The positioning offset.

DESCRIPTION

The PBRead function attempts to read ioReqCount bytes from the open file whose access path is specified in the ioRefNum field and transfer them to the data buffer pointed to by the ioBuffer field. The position of the mark is specified by ioPosMode and ioPosOffset. If your application tries to read past the logical end-of-file, PBRead reads the data, moves the mark to the end-of-file, and returns eofErr as its function result. Otherwise, PBRead moves the file mark to the byte following the last byte read and returns noErr. After the read is completed, the mark is returned in ioPosOffset, and the number of bytes actually read into the buffer is returned in ioActCount.

You can specify that PBRead read the file data 1 byte at a time until the requested number of bytes have been read or until the end-of-file is reached. To do so, set bit 7 of the ioPosMode field. Similarly, you can specify that PBRead should stop reading data when it reaches an application-defined newline character. To do so, place the ASCII code of that character into the high-order byte of the ioPosMode field; you must also set bit 7 of that field to enable newline mode.

Note

When reading data in newline mode, PBRead returns the newline character as part of the data read and sets ioActCount to the actual number of bytes placed into the buffer (which includes the newline character). ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBRead is _Read.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
fnOpnErr	–38	File not open
eofErr	–39	Logical end-of-file reached
posErr	–40	Attempt to position mark before start of file
fLckdErr	-45	File is locked
paramErr	–50	Negative ioReqCount
rfNumErr	–51	Bad reference number
afpAccessDenied	–5000	User does not have the correct access to the file

PBWrite

You can use the `PBWrite` function to write any number of bytes to an open file.

```
FUNCTION PBWrite (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioRefNum`	`Integer`	A file reference number.
→	`ioBuffer`	`Ptr`	A pointer to a data buffer.
→	`ioReqCount`	`LongInt`	The number of bytes requested.
←	`ioActCount`	`LongInt`	The number of bytes actually written.
→	`ioPosMode`	`Integer`	The positioning mode.
↔	`ioPosOffset`	`LongInt`	The positioning offset.

DESCRIPTION

The `PBWrite` function takes `ioReqCount` bytes from the buffer pointed to by `ioBuffer` and attempts to write them to the open file whose access path is specified by `ioRefNum`. The position of the mark is specified by `ioPosMode` and `ioPosOffset`. If the write operation completes successfully, `PBWrite` moves the file mark to the byte following the last byte written and returns `noErr`. After the write operation is completed, the mark is returned in `ioPosOffset` and the number of bytes actually written is returned in `ioActCount`.

If you try to write past the logical end-of-file, `PBWrite` moves the logical end-of-file. If you try to write past the physical end-of-file, `PBWrite` adds one or more clumps to the file and moves the physical end-of-file accordingly.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBWrite` is `_Write`.

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Disk full
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`posErr`	–40	Attempt to position mark before start of file
`wPrErr`	–44	Hardware volume lock
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Software volume lock

paramErr	–50	Negative ioReqCount
rfNumErr	–51	Bad reference number
wrPermErr	–61	Read / write permission doesn't allow writing

PBClose

You can use the PBClose function to close an open file.

```
FUNCTION PBClose (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
```

paramBlock A pointer to a basic File Manager parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioRefNum	Integer	A file reference number.

DESCRIPTION

The PBClose function writes the contents of the access path buffer specified by the ioRefNum field to the volume and removes the access path.

▲ **WARNING**
Some information stored on the volume won't be updated until PBFlushVol is called. ▲

▲ **WARNING**
Do not call PBClose with a file reference number of a file that has already been closed. Attempting to close the same file twice may result in loss of data on a volume. See "File Control Blocks" on page 2-82 for a description of how this can happen. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBClose is _Close.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
fnOpnErr	–38	File not open
fnfErr	–43	File not found
rfNumErr	–51	Bad reference number

Manipulating the File Mark

You can use the functions `PBGetFPos` and `PBSetFPos` to get or set the current position of the file mark.

PBGetFPos

You can use the `PBGetFPos` function to determine the current position of the file mark before reading from or writing to an open file.

```
FUNCTION PBGetFPos (paramBlock: ParmBlkPtr; async: Boolean):
                    OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioRefNum`	`Integer`	A file reference number.
←	`ioReqCount`	`LongInt`	On output, set to 0.
←	`ioActCount`	`LongInt`	On output, set to 0.
←	`ioPosMode`	`Integer`	On output, set to 0.
←	`ioPosOffset`	`LongInt`	The current position of the mark.

DESCRIPTION

The `PBGetFPos` function returns, in the `ioPosOffset` field, the mark of the specified file. The value returned in `ioPosOffset` is zero-based. Thus, a call to `PBGetFPos` returns 0 if you call it when the file mark is positioned at the beginning of the file.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBGetFPos` is `_GetFPos`.

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`rfNumErr`	–51	Bad reference number
`gfpErr`	–52	Error during `PBGetFPos`

PBSetFPos

You can use the PBSetFPos function to position the file mark before reading from or writing to an open file.

```
FUNCTION PBSetFPos (paramBlock: ParmBlkPtr; async: Boolean):
                    OSErr;
```

paramBlock A pointer to a basic File Manager parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioRefNum	Integer	A file reference number.
→	ioPosMode	Integer	The positioning mode.
↔	ioPosOffset	LongInt	On input, the positioning offset. On output, the position at which the mark was actually set.

DECRIPTION

The PBSetFPos function sets the mark of the specified file to the position specified by the ioPosMode and ioPosOffset fields. If you try to set the mark past the logical end-of-file, PBSetFPos moves the mark to the end-of-file and returns eofErr as its function result.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBSetFPos is _SetFPos.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
fnOpnErr	–38	File not open
eofErr	–39	Logical end-of-file reached
posErr	–40	Attempt to position mark before start of file
rfNumErr	–51	Bad reference number
extFSErr	–58	External file system

Manipulating the End-of-File

You can use the functions PBGetEOF and PBSetEOF to get or set the current end-of-file.

PBGetEOF

You can use the PBGetEOF function to determine the current logical end-of-file of an open file.

```
FUNCTION PBGetEOF (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
```

paramBlock A pointer to a basic File Manager parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioRefNum	Integer	A file reference number.
←	ioMisc	Ptr	The logical end-of-file.

DESCRIPTION

The PBGetEOF function returns, in the ioMisc field, the logical end-of-file of the specified file. Because ioMisc is of type Ptr, you'll need to coerce the value to type LongInt to interpret the value correctly.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBGetEOF is _GetEOF.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
fnOpnErr	–38	File not open
rfNumErr	–51	Bad reference number
afpAccessDenied	–5000	User does not have the correct access to the file

PBSetEOF

You can use the PBSetEOF function to set the logical end-of-file of an open file.

```
FUNCTION PBSetEOF (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
```

paramBlock A pointer to a basic File Manager parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioRefNum	Integer	A file reference number.
→	ioMisc	Ptr	The logical end-of-file.

DESCRIPTION

The PBSetEOF function sets the logical end-of-file of the open file, whose access path is specified by ioRefNum, to ioMisc. Because the ioMisc field is of type Ptr, you must coerce the desired value from type LongInt to type Ptr.

If you attempt to set the logical end-of-file beyond the current physical end-of-file, another allocation block is added to the file; if there isn't enough space on the volume, no change is made and PBSetEOF returns dskFulErr as its function result.

If the value of the ioMisc field is 0, all space occupied by the file on the volume is released. The file still exists, but it contains 0 bytes. Setting a file fork's end-of-file to 0 is therefore not the same as deleting the file (which removes both file forks at once).

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBSetEOF is _SetEOF.

RESULT CODES

noErr	0	No error
dskFulErr	–34	Disk full
ioErr	–36	I/O error
fnOpnErr	–38	File not open
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
rfNumErr	–51	Bad reference number
wrPermErr	–61	Read/write permission doesn't allow writing

Allocating File Blocks

The File Manager provides two low-level functions, PBAllocate and PBAllocContig, that allow you to allocate additional blocks to a file. The File Manager automatically allocates file blocks if you move the logical end-of-file past the physical end-of-file, and it automatically deallocates unneeded blocks from a file if you move the logical end-of-file to a position more than one allocation block before the current physical end-of-file. Consequently, you do not in general need to be concerned with allocating or deallocating file blocks. However, you can improve file block contiguity if you use the PBAllocate or PBAllocContig function to preallocate file blocks. This is most useful if you know in advance how big a file is likely to become.

PBAllocate and PBAllocContig are not supported by AppleShare volumes. Instead, use SetEOF or PBSetEOF to extend a file by setting the end-of-file.

PBAllocate

You can use the `PBAllocate` function to allocate additional blocks to an open file.

```
FUNCTION PBAllocate (paramBlock: ParmBlkPtr; async: Boolean):
                     OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioRefNum`	`Integer`	A file reference number.
→	`ioReqCount`	`LongInt`	The number of bytes requested.
←	`ioActCount`	`LongInt`	The number of bytes actually allocated, rounded up to the nearest multiple of the allocation block size.

DESCRIPTION

The `PBAllocate` function adds `ioReqCount` bytes to the specified file and sets the physical end-of-file to 1 byte beyond the last block allocated. If there isn't enough empty space on the volume to satisfy the allocation request, `PBAllocate` allocates the rest of the space on the volume and returns `dskFulErr` as its function result.

Note

If the total number of requested bytes is unavailable, `PBAllocate` allocates whatever space, contiguous or not, is available. To force the allocation of the entire requested space as a contiguous piece, call `PBAllocContig` instead. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBAllocate` is `_Allocate`.

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Disk full
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`wPrErr`	–44	Hardware volume lock
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Software volume lock
`rfNumErr`	–51	Bad reference number
`wrPermErr`	–61	Read/write permission doesn't allow writing

PBAllocContig

You can use the PBAllocContig function to allocate additional contiguous blocks to an open file.

```
FUNCTION PBAllocContig (paramBlock: ParmBlkPtr; async: Boolean):
                        OSErr;
```

paramBlock A pointer to a basic File Manager parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioRefNum	Integer	A file reference number.
→	ioReqCount	LongInt	The number of bytes requested.
←	ioActCount	LongInt	The number of bytes allocated, rounded up to the nearest multiple of the allocation block size.

DESCRIPTION

The PBAllocContig function is identical to the PBAllocate function except that if there isn't enough contiguous empty space on the volume to satisfy the allocation request, PBAllocContig does nothing and returns dskFulErr as its function result. If you want to allocate whatever space is available, even when the entire request cannot be filled by the allocation of a contiguous piece, call PBAllocate instead.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBAllocContig is _AllocContig.

RESULT CODES

noErr	0	No error
dskFulErr	–34	Disk full
ioErr	–36	I/O error
fnOpnErr	–38	File not open
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
rfNumErr	–51	Bad reference number
wrPermErr	–61	Read/write permission doesn't allow writing

Updating Files

You can use the `PBFlushFile` function to ensure that the path access buffer of a file is written to disk. There is no high-level equivalent of this function.

PBFlushFile

You can use the `PBFlushFile` function to write the contents of a file's access path buffer.

```
FUNCTION PBFlushFile (paramBlock: ParmBlkPtr; async: Boolean):
                        OSErr;
```

`paramBlock` A pointer to a basic FIle Manager parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioRefNum`	`Integer`	A file reference number.

DESCRIPTION

The `PBFlushFile` function writes the contents of the access path buffer indicated by `ioRefNum` to the volume and then updates the file's entry in the volume catalog.

▲ **WARNING**
Some information stored on the volume won't be correct until `PBFlushVol` is called. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBFlushFile` is `_FlushFile`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`fnfErr`	–43	File not found
`rfNumErr`	–51	Bad reference number
`extFSErr`	–58	External file system

High-Level Volume Access Routines

This section describes the File Manager's high-level routines for accessing volumes. Most applications are likely to need only the `FlushVol` function described on page 2-135.

When you call one of these routines, you specify a volume by a volume reference number (which you can obtain, for example, by calling the `GetVInfo` function, or from the reply record returned by the Standard File Package). You can also specify a volume by name, but this is generally discouraged, because there is no guarantee that volume names will be unique.

Unmounting Volumes

The functions `UnmountVol` and `Eject` allow you to unmount and eject volumes. Most applications do not need to use these routines, because the user typically ejects (and possibly also unmounts) a volume in the Finder.

UnmountVol

You can use the `UnmountVol` function to unmount a volume that isn't currently being used.

```
FUNCTION UnmountVol (volName: StringPtr; vRefNum: Integer): OSErr;
```

`volName` A pointer to the name of a mounted volume.

`vRefNum` A volume reference number, a working directory reference number, a drive number, or 0 for the default volume.

DESCRIPTION

The `UnmountVol` function unmounts the specified volume. All files on the volume (except those opened by the Operating System) must be closed before you call `UnmountVol`, which does not eject the volume.

▲ **WARNING**

Don't unmount the startup volume. Doing so will cause a system crash. ▲

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad volume name
fBsyErr	–47	One or more files are open
paramErr	–50	No default volume
nsDrvErr	–56	No such drive
extFSErr	–58	External file system

Eject

You can use the `Eject` function to place a volume offline and eject it.

```
FUNCTION Eject (volName: StringPtr; vRefNum: Integer): OSErr;
```

`volName` A pointer to the name of a volume.

`vRefNum` A volume reference number, a working directory reference number, a drive number, or 0 for the default volume.

DESCRIPTION

The `Eject` function flushes the specified volume, places it offline, and then ejects the volume.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad volume name
paramErr	–50	No default volume
nsDrvErr	–56	No such drive
extFSErr	–58	External file system

Updating Volumes

When you close a file, you should call `FlushVol` to ensure that any changed contents of the file are written to the volume.

FlushVol

You can use the `FlushVol` function to write the contents of the volume buffer and update information about the volume.

```
FUNCTION FlushVol (volName: StringPtr; vRefNum: Integer): OSErr;
```

`volName` A pointer to the name of a mounted volume.

`vRefNum` A volume reference number, a working directory reference number, a drive number, or 0 for the default volume.

DESCRIPTION

On the specified volume, the `FlushVol` function writes the contents of the associated volume buffer and descriptive information about the volume (if they've changed since the last time `FlushVol` was called). This information is written to the volume.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad volume name
`paramErr`	–50	No default volume
`nsDrvErr`	–56	No such drive

Manipulating the Default Volume

The functions `GetVol`, `SetVol`, `HGetVol`, and `HSetVol` allow you to determine which volume is the default volume and to set the default volume.

GetVol

You can use the `GetVol` function to determine the current default volume and possibly also the default directory.

```
FUNCTION GetVol (volName: StringPtr; VAR vRefNum: Integer): OSErr;
```

`volName` A pointer to the name of the default volume.

`vRefNum` A volume reference number or a working directory reference number.

DESCRIPTION

The `GetVol` function returns a pointer to the name of the default volume in the `volName` parameter and its volume reference number in the `vRefNum` parameter. If the default directory has a working directory associated with it, the `vRefNum` parameter instead contains a working directory reference number (which encodes both the volume reference number and the default directory ID). However, if, in a previous call to `HSetVol` (or `PBHSetVol`), a working directory reference number was passed in, `GetVol` returns a volume reference number in the `vRefNum` parameter.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume

SetVol

You can change the default volume and default directory using the `SetVol` function.

```
FUNCTION SetVol (volName: StringPtr; vRefNum: Integer): OSErr;
```

`volName` A pointer to the name of a mounted volume.

`vRefNum` A volume reference number or a working directory reference number.

DESCRIPTION

The `SetVol` function sets the default volume and directory to the values specified in the `volName` and `vRefNum` parameters. If you pass a volume reference number in `vRefNum` or a pointer to a volume name in `volName`, `SetVol` makes the specified volume the default volume and the root directory of that volume the default directory. If you pass a working directory reference number in `vRefNum`, `SetVol` makes the specified directory the default directory, and the volume containing that directory the default volume.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`bdNamErr`	–37	Bad volume name
`paramErr`	–50	No default volume

HGetVol

You can use the `HGetVol` function to determine the current default volume and default directory.

```
FUNCTION HGetVol (volName: StringPtr; VAR vRefNum: Integer;
                  VAR dirID: LongInt): OSErr;
```

`volName` A pointer to the name of the default volume.

`vRefNum` A volume reference number or a working directory reference number.

`dirID` The directory ID of the default directory.

DESCRIPTION

The `HGetVol` function returns the name and reference number of the default volume, as well as the directory ID of the default directory. A pointer to the name of the default volume is returned in the `volName` parameter, unless you set `volName` to `NIL` before calling `HGetVol`.

The `HGetVol` function returns a working directory reference number in the `vRefNum` parameter if the previous call to `HSetVol` (or `PBHSetVol`) passed in a working directory reference number. If, however, you have previously called `HSetVol` (or `PBHSetVol`) specifying the target volume with a volume reference number, then `HGetVol` returns a volume reference number in the `vRefNum` parameter.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No default volume

HSetVol

You can use the `HSetVol` function to set both the default volume and the default directory.

```
FUNCTION HSetVol (volName: StringPtr; vRefNum: Integer;
                  dirID: LongInt): OSErr;
```

`volName` A pointer to the name of a mounted volume or the partial pathname of a directory.

`vRefNum` A volume reference number or a working directory reference number.

`dirID` A directory ID.

DESCRIPTION

The `HSetVol` function lets you specify the default directory by volume reference number, by directory ID, or by a combination of working directory reference number and partial pathname (beginning from that working directory).

▲ **WARNING**

Use of the `HSetVol` function is discouraged if your application may execute in system software versions prior to version 7.0. Because the specified directory might not itself be a working directory, `HSetVol` records the default volume and directory separately, using the volume reference number of the volume and the actual directory ID of the specified directory. Subsequent calls to `GetVol` (or `PBGetVol`) return only the volume reference number, which will cause that volume's root directory (rather than the default directory, as expected) to be accessed. ▲

Note

Both the default volume and the default directory are used in calls made with no volume name, a volume reference number of 0, and a directory ID of 0. ◆

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`bdNamErr`	–37	Bad volume name
`fnfErr`	–43	Directory not found
`paramErr`	–50	No default volume
`afpAccessDenied`	–5000	User does not have access to the directory

Obtaining Volume Information

You can get information about a volume by calling the `GetVInfo` or `GetVRefNum` function.

GetVInfo

You can use the `GetVInfo` function to get information about a mounted volume.

```
FUNCTION GetVInfo (drvNum: Integer; volName: StringPtr;
                   VAR vRefNum: Integer;
                   VAR freeBytes: LongInt): OSErr;
```

`drvNum`	The drive number of the volume for which information is requested.
`volName`	On output, a pointer to the name of the specified volume.
`vRefNum`	The volume reference number of the specified volume.
`freeBytes`	The available space (in bytes) on the specified volume.

DESCRIPTION

The `GetVInfo` function returns the name, volume reference number, and available space (in bytes) for the specified volume. You specify a volume by providing its drive number in the `drvNum` parameter. You can pass 0 in the `drvNum` parameter to get information about the default volume.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`paramErr`	–50	No default volume

GetVRefNum

You can use the `GetVRefNum` function to get a volume reference number from a file reference number.

```
FUNCTION GetVRefNum (refNum: Integer; VAR vRefNum: Integer):
                     OSErr;
```

`refNum` The file reference number of an open file.

`vRefNum` On exit, the volume reference number of the volume containing the file specified by `refNum`.

DESCRIPTION

The `GetVRefNum` function returns the volume reference number of the volume containing the specified file. If you also want to determine the directory ID of the specified file's parent directory, call the `PBGetFCBInfo` function.

RESULT CODES

`noErr`	0	No error
`rfNumErr`	–51	Bad reference number

Low-Level Volume Access Routines

This section describes the low-level routines for accessing volumes. These routines exchange parameters with your application through a parameter block of type `ParamBlock`, `HParamBlock`, or `WDPBRec`. When you call a low-level routine, you pass the address of the appropriate parameter block to the routine.

Some low-level routines for accessing volumes can run either asynchronously or synchronously. Each of these routines comes in three versions: one version requires the `async` parameter and two have the suffix `Async` or `Sync` added to their names. For

more information about the differences between the three versions, see “Low-Level File Access Routines” on page 2-121.

Only the first version of these routines is documented in this section. See “Summary of the File Manager,” beginning on page 2-243, for a listing that includes all three versions.

Assembly-Language Note

See the assembly-language note on page 2-121 for details on calling these routines from assembly language. ◆

Mounting and Unmounting Volumes

The File Manager provides several low-level routines that allow you to mount and unmount Macintosh volumes, eject volumes, and place mounted volumes offline.

PBMountVol

You can use the `PBMountVol` function to mount a volume.

```
FUNCTION PBMountVol (paramBlock: ParmBlkPtr): OSErr;
```

`paramBlock` A pointer to a basic FIle Manager parameter block.

Parameter block

←	`ioResult`	`OSErr`	The result code of the function.
↔	`ioVRefNum`	`Integer`	On input, a drive number. On output, the volume reference number.

DESCRIPTION

The `PBMountVol` function mounts the volume in the specified drive. If there are no volumes already mounted, this volume becomes the default volume.

Because you specify the volume to be mounted by providing a drive number, you can use `PBMountVol` to mount only one volume per disk.

The `PBMountVol` function always executes synchronously.

Note

The `PBMountVol` function opens two files needed for maintaining file catalog and file mapping information. If no access paths are available for these two files, `PBMountVol` fails and returns `tmfoErr` as its function result. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBMountVol` is `_MountVol`.

RESULT CODES

noErr	0	No error
ioErr	–36	I/O error
tmfoErr	–42	Too many files open
paramErr	–50	Bad drive number
volOnLinErr	–55	Volume already online
nsDrvErr	–56	No such drive
noMacDskErr	–57	Not a Macintosh disk
extFSErr	–58	External file system
badMDBErr	–60	Bad master directory block
memFullErr	–108	Not enough room in heap zone

PBUnmountVol

You can use the PBUnmountVol function to unmount a volume.

```
FUNCTION PBUnmountVol (paramBlock: ParmBlkPtr): OSErr;
```

paramBlock A pointer to a basic File Manager parameter block.

Parameter block

←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume reference number, a working directory reference number, or 0 for the default volume.

DESCRIPTION

The PBUnmountVol function unmounts the specified volume. All user files on the volume must be closed. Then, PBUnmountVol calls PBFlushVol to flush the volume and releases the memory used for the volume.

The PBUnmountVol function always executes synchronously.

▲ **WARNING**

Don't unmount the startup volume. Doing so will cause a system crash. ▲

Note

Unmounting a volume does not close working directories; to release the memory allocated to a working directory, call PBCloseWD. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBUnmountVol is _UnmountVol.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad volume name
fBsyErr	–47	One or more files are open
paramErr	–50	No default volume
nsDrvErr	–56	No such drive
extFSErr	–58	External file system

PBEject

When your application is finished with a volume, you can use the `PBEject` function to place the volume offline and eject it.

```
FUNCTION PBEject (paramBlock: ParmBlkPtr): OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.

DESCRIPTION

The `PBEject` function flushes the specified volume, places it offline, and then ejects the volume.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBEject` is `_Eject`. You can invoke the `_Eject` macro asynchronously; the first two parts of the call are executed synchronously, and the actual ejection is executed asynchronously.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad volume name
paramErr	–50	No default volume
nsDrvErr	–56	No such drive
extFSErr	–58	External file system

PBOffLine

You can use the `PBOffLine` function to place a volume offline. Most applications don't need to do this.

```
FUNCTION PBOffLine (paramBlock: ParmBlkPtr): OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.

DESCRIPTION

The `PBOffLine` function places the specified volume offline by calling `PBFlushVol` to flush the volume and releasing all the memory used for the volume except for the volume control block.

The `PBOffLine` function always executes synchronously.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBOffLine` is `_OffLine`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad volume name
`paramErr`	–50	No default volume
`nsDrvErr`	–56	No such drive
`extFSErr`	–58	External file system

Updating Volumes

You can update a volume by calling the `PBFlushVol` function.

PBFlushVol

You can use the `PBFlushVol` function to write the contents of the volume buffer and update information about the volume.

```
FUNCTION PBFlushVol (paramBlock: ParmBlkPtr; async: Boolean):
                     OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.

DESCRIPTION

On the volume specified by `ioNamePtr` or `ioVRefNum`, the `PBFlushVol` function writes descriptive information about the volume, the contents of the associated volume buffer, and all access path buffers for the volume (if they've changed since the last time `PBFlushVol` was called).

Note

The date and time of the last modification to the volume are set when the modification is made, not when the volume is flushed. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBFlushVol` is `_FlushVol`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad volume name
`paramErr`	–50	No default volume
`nsDrvErr`	–56	No such drive
`extFSErr`	–58	External file system

Obtaining Volume Information

The File Manager provides several routines that allow you to obtain and modify information about a volume. For example, you can use the `PBHGetVInfo` function to determine the date and time that a volume was last modified. You can use the `PBHGetVolParms` function to determine other features of the volume, such as whether it supports the `PBHOpenDeny` function.

PBHGetVInfo

You can use the `PBHGetVInfo` function to get detailed information about a volume.

```
FUNCTION PBHGetVInfo (paramBlock: HParmBlkPtr; async: Boolean):
                        OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
↔	`ioNamePtr`	`StringPtr`	A pointer to the volume's name.
↔	`ioVRefNum`	`Integer`	On input, a volume specification. On output, the volume reference number.
→	`ioVolIndex`	`Integer`	An index used for indexing through all mounted volumes.
←	`ioVCrDate`	`LongInt`	The date and time of initialization.
←	`ioVLsMod`	`LongInt`	The date and time of last modification.
←	`ioVAtrb`	`Integer`	The volume attributes.
←	`ioVNmFls`	`Integer`	The number of files in the root directory.
←	`ioVBitMap`	`Integer`	The first block of the volume bitmap.
←	`ioVAllocPtr`	`Integer`	The block at which the next new file starts.
←	`ioVNmAlBlks`	`Integer`	The number of allocation blocks.
←	`ioVAlBlkSiz`	`LongInt`	The size of allocation blocks.
←	`ioVClpSiz`	`LongInt`	The default clump size.
←	`ioAlBlSt`	`Integer`	The first block in the volume block map.
←	`ioVNxtCNID`	`LongInt`	The next unused catalog node ID.
←	`ioVFrBlk`	`Integer`	The number of unused allocation blocks.
←	`ioVSigWord`	`Integer`	The volume signature.
←	`ioVDrvInfo`	`Integer`	The drive number.
←	`ioVDRefNum`	`Integer`	The driver reference number.

←	ioVFSID	Integer	The file system handling this volume.
←	ioVBkUp	LongInt	The date and time of the last backup.
←	ioVSeqNum	Integer	Used internally.
←	ioVWrCnt	LongInt	The volume write count.
←	ioVFilCnt	LongInt	The number of files on the volume.
←	ioVDirCnt	LongInt	The number of directories on the volume.
←	ioVFndrInfo	ARRAY[1..8] OF LongInt	Information used by the Finder.

DESCRIPTION

The PBHGetVInfo function returns information about the specified volume. If the value of ioVolIndex is positive, the File Manager attempts to use it to find the volume; for instance, if the value of ioVolIndex is 2, the File Manager attempts to access the second mounted volume in the VCB queue. If the value of ioVolIndex is negative, the File Manager uses ioNamePtr and ioVRefNum in the standard way to determine the volume. If the value of ioVolIndex is 0, the File Manager attempts to access the volume by using ioVRefNum only. The volume reference number is returned in ioVRefNum, and the volume name is returned in the buffer whose address you passed in ioNamePtr. You should pass a pointer to a Str31 value if you want that name returned. If you pass NIL in the ioNamePtr field, no volume name is returned.

If you pass a working directory reference number in ioVRefNum (or if the default directory is a subdirectory), the number of files and directories in the specified directory (the directory's valence) is returned in ioVNmFls.

You can read the ioVDrvInfo and ioVDRefNum fields to determine whether the specified volume is online, offline, or ejected. For online volumes, ioVDrvInfo contains the drive number of the drive containing the specified volume and hence is always greater than 0. If the value returned in ioVDrvInfo is 0, the volume is either offline or ejected. You can determine whether the volume is offline or ejected by inspecting the value of the ioVDRefNum field. For online volumes, ioVDRefNum contains a driver reference number; these numbers are always less than 0. If the volume is not online, the value of ioVDRefNum is either the negative of the drive number (if the volume is offline) or the drive number itself (if the volume is ejected).

You can get information about all the online volumes by making repeated calls to PBHGetVInfo, starting with the value of ioVolIndex set to 1 and incrementing that value until PBHGetVInfo returns nsvErr.

SPECIAL CONSIDERATIONS

The values returned in the ioVNmAlBlks and ioVFrBlk fields are unsigned integers. You need to exercise special care when reading those values from Pascal. See "Determining the Amount of Free Space on a Volume" on page 2-47 for one technique you can use to read those values.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBHGetVInfo` is `_HGetVolInfo`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`paramErr`	–50	No default volume

PBSetVInfo

You can use the `PBSetVInfo` function to change information about a volume.

```
FUNCTION PBSetVInfo (paramBlock: HParmBlkPtr; async: Boolean):
                     OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to the volume's name.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioVCrDate`	`LongInt`	The date and time of initialization.
→	`ioVLsMod`	`LongInt`	The date and time of last modification.
→	`ioVAtrb`	`Integer`	The volume attributes.
→	`ioVBkUp`	`LongInt`	The date and time of the last backup.
→	`ioVSeqNum`	`Integer`	Used internally.
→	`ioVFndrInfo`	`ARRAY[1..8] OF LongInt`	Information used by the Finder.

DESCRIPTION

The `PBSetVInfo` function lets you modify information about volumes. You can specify, in `ioNamePtr`, a pointer to a new name for the volume. Only bit 15 of `ioVAtrb` can be changed; setting it locks the volume.

Note
You cannot specify the volume by name; you must use either the volume reference number, the drive number, or a working directory reference number. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBSetVInfo` is `_SetVolInfo`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`paramErr`	–50	No default volume

PBHGetVolParms

You can use the `PBHGetVolParms` function to determine the characteristics of a volume.

```
FUNCTION PBHGetVolParms (paramBlock: HParmBlkPtr; async: Boolean):
                         OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to the volume's name.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioBuffer`	`Ptr`	A pointer to a `GetVolParmsInfoBuffer` record.
→	`ioReqCount`	`LongInt`	The size of the buffer area.
←	`ioActCount`	`LongInt`	The size of the data actually returned.

DESCRIPTION

The `PBHGetVolParms` function returns information about the characteristics of a volume. You specify a volume (either by name or by volume reference number) and a buffer size, and `PBHGetVolParms` fills in the volume attributes buffer, as described in this section.

You can use a name (pointed to by the `ioNamePtr` field) or a volume specification (contained in the `ioVRefNum` field) to specify the volume. A volume specification can be a volume reference number, drive number, or working directory reference number. If you use a volume specification to specify the volume, you should set the `ioNamePtr` field to `NIL`.

You must allocate memory to hold the returned attributes and put a pointer to the buffer in the `ioBuffer` field. Specify the size of the buffer in the `ioReqCount` field. The `PBHGetVolParms` function places the attributes information in the buffer pointed to by the `ioBuffer` field and specifies the actual length of the data in the `ioActCount` field.

The PBHGetVolParms function returns the bulk of its volume description in the vMAttrib field of the attributes buffer. The vMAttrib field contains 32 bits of attribute information about the volume. Bits 0–3 and 21–24 are reserved; all volumes should return these bits clear. The bits currently used are defined by these constants:

```
CONST
   bHasBlankAccessPrivileges
                  =  4;    {volume supports inherited privileges}
   bHasBTreeMgr   =  5;    {reserved}
   bHasFileIDs    =  6;    {volume supports file ID functions}
   bHasCatSearch  =  7;    {volume supports PBCatSearch}
   bHasUserGroupList
                  =  8;    {volume supports AFP privileges}
   bHasPersonalAccessPrivileges
                  =  9;    {local file sharing is enabled}
   bHasFolderLock =  10;   {volume supports locking of folders}
   bHasShortName  =  11;   {volume supports AFP short names}
   bHasDesktopMgr =  12;   {volume supports Desktop Manager}
   bHasMoveRename =  13;   {volume supports _MoveRename}
   bHasCopyFile   =  14;   {volume supports _CopyFile}
   bHasOpenDeny   =  15;   {volume supports shared access modes}
   bHasExtFSVol   =  16;   {volume is external file system volume}
   bNoSysDir      =  17;   {volume has no system directory}
   bAccessCntl    =  18;   {volume supports AFP access control}
   bNoBootBlks    =  19;   {volume is not a startup volume}
   bNoDeskItems   =  20;   {do not place objects on the desktop}
   bNoSwitchTo    =  25;   {do not switch launch to applications}
   bTrshOffLine   =  26;   {zoom volume when it is unmounted}
   bNoLclSync     =  27;   {don't let Finder change mod. date}
   bNoVNEdit      =  28;   {lock volume name}
   bNoMiniFndr    =  29;   {reserved; always 1}
   bLocalWList    =  30;   {use shared volume handle for window }
                           { list}
   bLimitFCBs     =  31;   {limit file control blocks}
```

These constants have the following meanings if set:

Constant descriptions

`bHasBlankAccessPrivileges`
: This volume supports inherited access privileges for folders.

`bHasBTreeMgr`
: Reserved for internal use.

`bHasFileIDs`
: This volume supports the file ID functions, including the `PBExchangeFiles` function.

`bHasCatSearch`
: This volume supports the `PBCatSearch` function.

`bHasUserGroupList`
: This volume supports the Users and Groups file and thus the AFP privilege functions.

`bHasPersonalAccessPrivileges`
: This volume has local file sharing enabled.

`bHasFolderLock` Folders on the volume can be locked, and so they cannot be deleted or renamed.

`bHasShortName` This volume supports AFP short names.

`bHasDesktopMgr` This volume supports all of the desktop functions (described in the chapter "Desktop Manager" in *Inside Macintosh: More Macintosh Toolbox*).

`bHasMoveRename` This volume supports the `PBHMoveRename` function.

`bHasCopyFile` This volume supports the `PBHCopyFile` function, which is used in copy and duplicate operations if both source and destination volumes have the same server address.

`bHasOpenDeny` This volume supports the `PBHOpenDeny` and `PBHOpenRFDeny` functions.

`bHasExtFSVol` This volume is an external file system volume.

`bNoSysDir` This volume doesn't support a system directory. Do not switch launch to this volume.

`bAccessCntl` This volume supports AppleTalk AFP access-control interfaces. The `PBHGetLoginInfo`, `PBHGetDirAccess`, `PBHSetDirAccess`, `PBHMapID`, and `PBHMapName` functions are supported. Special folder icons are used. The Access Privileges menu command is enabled for disk and folder items. The `ioFlAttrib` field of `PBGetCatInfo` calls is assumed to be valid.

`bNoBootBlks` This volume is not a startup volume. The Startup menu item is disabled. Boot blocks are not copied during copy operations.

`bNoDeskItems` Don't place objects in this volume on the Finder desktop.

`bNoSwitchTo` The Finder will not switch launch to any application on this volume.

`bTrshOffLine` Any time this volume goes offline, it is zoomed to the Trash and unmounted.

`bNoLclSync` Don't let the Finder change the modification date.

`bNoVNEdit` This volume's name cannot be edited.

`bNoMiniFndr` Reserved; always set to 1.

`bLocalWList` The Finder uses the returned shared volume handle for its local window list.

`bLimitFCBs` The Finder limits the number of file control blocks used during copying to 8 instead of 16.

SPECIAL CONSIDERATIONS

A volume's characteristics can change when the user enables and disables file sharing. You might have to make repeated calls to `PBHGetVolParms` to ensure that you have the current status of a volume.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBHGetVolParms` are

Trap macro	Selector
`_HFSDispatch`	$0030

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`paramErr`	–50	Volume doesn't support the function

Manipulating the Default Volume

The low-level functions `PBGetVol`, `PBSetVol`, `PBHGetVol`, and `PBHSetVol` allow you to manipulate the default volume and directory.

PBGetVol

You can use the `PBGetVol` function to determine the default volume and default directory.

```
FUNCTION PBGetVol (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
←	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
←	`ioVRefNum`	`Integer`	A volume reference number or a working directory reference number.

DESCRIPTION

The `PBGetVol` function returns, in `ioNamePtr`, a pointer to the name of the default volume (unless `ioNamePtr` is `NIL`) and, in `ioVRefNum`, its volume reference number. If a default directory was set with a previous call to `PBSetVol`, a pointer to its name is returned in `ioNamePtr` and its working directory reference number is returned in `ioVRefNum`. However, if, in a previous call to `HSetVol` (or `PBHSetVol`), a working directory reference number was passed in, `PBGetVol` returns a volume reference number in the `ioVRefNum` field.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBGetVol` is `_GetVol`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No default volume

PBSetVol

You can change the default volume and default directory using the `PBSetVol` function.

```
FUNCTION PBSetVol (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume reference number or a working directory reference number.

DESCRIPTION

If you pass a volume reference number in `ioVRefNum`, the `PBSetVol` function makes the specified volume the default volume and the root directory of that volume the default directory. If you pass a working directory reference number, `PBSetVol` makes the specified directory the default directory, and the volume containing that directory the default volume.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBSetVol` is `_SetVol`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`bdNamErr`	–37	Bad volume name
`paramErr`	–50	No default volume

PBHGetVol

You can use the PBHGetVol function to determine the default volume and default directory.

```
FUNCTION PBHGetVol (paramBlock: WDPBPtr; async: Boolean): OSErr;
```

paramBlock A pointer to a working directory parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
←	ioNamePtr	StringPtr	A pointer to a pathname.
←	ioVRefNum	Integer	A volume reference number or a working directory reference number.
←	ioWDProcID	LongInt	The working directory user identifier.
←	ioWDVRefNum	Integer	The volume reference number of the default volume.
←	ioWDDirID	LongInt	The directory ID of the default directory.

DESCRIPTION

The PBHGetVol function returns the default volume and directory last set by a call to either PBSetVol or PBHSetVol. The reference number of the default volume is returned in ioVRefNum. The PBHGetVol function returns a pointer to the volume's name in the ioNamePtr field. You should pass a pointer to a Str31 value if you want that name returned. If you pass NIL in the ioNamePtr field, no volume name is returned.

▲ **WARNING**
On exit, the ioVRefNum field contains a working directory reference number (instead of the volume reference number) if, in the last call to PBSetVol or PBHSetVol, a working directory reference number was passed in this field. ▲

The volume reference number of the volume on which the default directory exists is returned in ioWDVRefNum. The directory ID of the default directory is returned in ioWDDirID.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBHGetVol is _HGetVol.

RESULT CODES

noErr	0	No error
nsvErr	–35	No default volume

PBHSetVol

The `PBHSetVol` function sets both the default volume and the default directory.

```
FUNCTION PBHSetVol (paramBlock: WDPBPtr; async: Boolean): OSErr;
```

`paramBlock` A pointer to a working directory parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume reference number or a working directory reference number.
→	`ioWDDirID`	`LongInt`	The directory ID.

DESCRIPTION

The `PBHSetVol` function sets the default volume and directory to the volume and directory specified by the `ioNamePtr`, `ioVRefNum`, and `ioWDDirID` fields.

The `PBHSetVol` function sets the default volume to the volume specified by the `ioVRefNum` field, which can contain either a volume reference number or a working directory reference number. If the `ioNamePtr` field specifies a full pathname, however, the default volume is set to the volume whose name is contained in that pathname. (A full pathname overrides the `ioVRefNum` field.)

The `PBHSetVol` function also sets the default directory. If the `ioVRefNum` field contains a volume reference number, then the default directory is set to the directory on that volume having the partial pathname specified by `ioNamePtr` in the directory specified by `ioWDDirID`. If the value of `ioNamePtr` is `NIL`, the default directory is simply the directory whose directory ID is contained in `ioWDDirID`.

If the `ioVRefNum` field contains a working directory reference number, then `ioWDDirID` is ignored and the default directory is set to the directory on that volume having the partial pathname specified by `ioNamePtr` in the directory specified by the working directory reference number. If the value of `ioNamePtr` is `NIL`, the default directory is simply the directory specified in `ioVRefNum`.

▲ **WARNING**

Use of the `PBHSetVol` function is discouraged if your application may execute in system software versions prior to version 7.0. Because the specified directory might not itself be a working directory, `PBHSetVol` records the default volume and directory separately, using the volume reference number of the volume and the actual directory ID of the specified directory. Subsequent calls to `GetVol` (or `PBGetVol`) return only the volume reference number, which will cause that volume's root directory (rather than the default directory, as expected) to be accessed. ▲

Note

Both the default volume *and* the default directory are used in calls made with no volume name, a volume reference number of 0, and a directory ID of 0. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBSGetVol` is `_HSetVol`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`bdNamErr`	–37	Bad volume name
`fnfErr`	–43	Directory not found
`paramErr`	–50	No default volume
`afpAccessDenied`	–5000	User does not have access to the directory

File System Specification Routines

The File Manager provides a set of file and directory manipulation routines that accept file system specification records as parameters. Depending on the requirements of your application and on the environment in which it is running, you may be able to accomplish all your file and directory operations by using these routines.

Before calling any of these routines, however, you should call the `Gestalt` function to ensure that they are available in the operating environment. If these routines are not available, you can call the corresponding HFS routines. See "High-Level HFS Routines" on page 2-170 for details.

Opening Files

There are two `FSSpec` functions that allow you to open files, `FSpOpenDF` and `FSpOpenRF`. You can use them to open a file's data fork and resource fork, respectively.

FSpOpenDF

You can use the `FSpOpenDF` function to open a file's data fork.

```
FUNCTION FSpOpenDF (spec: FSSpec; permission: SignedByte;
                    VAR refNum: Integer): OSErr;
```

`spec`	An `FSSpec` record specifying the file whose data fork is to be opened.
`permission`	A constant indicating the desired file access permissions.
`refNum`	A reference number of an access path to the file's data fork.

DESCRIPTION

The FSpOpenDF function opens the data fork of the file specified by the spec parameter and returns a file reference number in the refNum parameter. You can pass that reference number as a parameter to any of the low- or high-level file access routines.

The permission parameter specifies the kind of access permission mode you want. In most cases, you can simply set the permission parameter to fsCurPerm. Some applications request fsRdWrPerm, to ensure that they can both read from and write to a file. For more information about permissions, see "File Manipulation" on page 2-7. In shared environments, permission requests are translated into the deny mode permissions defined by AppleShare.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for FSpOpenDF are

Trap macro	Selector
_HighLevelHFSDispatch	$0002

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
tmfoErr	–42	Too many files open
fnfErr	–43	File not found
opWrErr	–49	File already open for writing
permErr	–54	Attempt to open locked file for writing
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file

FSpOpenRF

You can use the FSpOpenRF function to open a file's resource fork.

```
FUNCTION FSpOpenRF (spec: FSSpec; permission: SignedByte;
                    VAR refNum: Integer): OSErr;
```

spec	An FSSpec record specifying the file whose resource fork is to be opened.
permission	A constant indicating the desired file access permissions.
refNum	A reference number of an access path to the file's resource fork.

DESCRIPTION

The FSpOpenRF function creates an access path to the resource fork of a file and returns, in the refNum parameter, an access path reference number to that fork. You can pass that

reference number as a parameter to any of the low- or high-level file access routines. The `permission` parameter should contain a constant indicating the desired file access permissions.

SPECIAL CONSIDERATIONS

Generally, your application should use Resource Manager routines rather than File Manager routines to access a file's resource fork. The `FSpOpenRF` function does not read the resource map into memory and is generally useful only for applications (such as utilities that copy files) that need block-level access to a resource fork. In particular, you should not use the resource fork of a file to hold nonresource data. Many parts of the system software assume that a resource fork always contains resource data.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `FSpOpenRF` are

Trap macro	Selector
`_HighLevelHFSDispatch`	$0003

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`tmfoErr`	–42	Too many files open
`fnfErr`	–43	File not found
`opWrErr`	–49	File already open for writing
`permErr`	–54	Attempt to open locked file for writing
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access to the file

Creating and Deleting Files and Directories

You can create files and directories by calling `FSpCreate` and `FSpDirCreate`, respectively. You can delete files and directories by calling the `FSpDelete` function.

FSpCreate

You can use the `FSpCreate` function to create a new file.

```
FUNCTION FSpCreate (spec: FSSpec; creator: OSType;
                    fileType: OSType; scriptTag: ScriptCode):
                    OSErr;
```

`spec` An `FSSpec` record specifying the file to be created.

`creator`	The creator of the new file.
`fileType`	The file type of the new file.
`scriptTag`	The code of the script system in which the filename is to be displayed. If you have established the name and location of the new file using either the `StandardPutFile` or `CustomPutFile` procedure, specify the script code returned in the reply record. (See the chapter "Standard File Package" in this book for a description of `StandardPutFile` and `CustomPutFile`.) Otherwise, specify the system script by setting the `scriptTag` parameter to the value `smSystemScript`.

DESCRIPTION

The `FSpCreate` function creates a new file (both forks) with the specified type, creator, and script code. The new file is unlocked and empty. The date and time of creation and last modification are set to the current date and time.

See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on file types and creators.

Files created using `FSpCreate` are not automatically opened. If you want to write data to the new file, you must first open the file using a file access routine (such as `FSpOpenDF`).

Note

The resource fork of the new file exists but is empty. You'll need to call one of the Resource Manager procedures `CreateResFile`, `HCreateResFile`, or `FSpCreateResFile` to create a resource map in the file before you can open it (by calling one of the Resource Manager functions `OpenResFile`, `HOpenResFile`, or `FSpOpenResFile`). ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `FSpCreate` are

Trap macro	Selector
`_HighLevelHFSDispatch`	$0004

RESULT CODES

`noErr`	0	No error
`dirFulErr`	–33	File directory full
`dskFulErr`	–34	Disk is full
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	Directory not found or incomplete pathname
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`dupFNErr`	–48	Duplicate filename and version
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access
`afpObjectTypeErr`	–5025	A directory exists with that name

FSpDirCreate

You can use the `FSpDirCreate` function to create a new directory.

```
FUNCTION FSpDirCreate (spec: FSSpec; scriptTag: ScriptCode;
                       VAR createdDirID: LongInt): OSErr;
```

`spec` An `FSSpec` record specifying the directory to be created.

`scriptTag` The code of the script system in which the directory name is to be displayed. If you have established the name and location of the new directory using either the `StandardPutFile` or `CustomPutFile` procedure, specify the script code returned in the reply record. (See the chapter "Standard File Package" in this book for a description of `StandardPutFile` and `CustomPutFile`.) Otherwise, specify the system script by setting the `scriptTag` parameter to the value `smSystemScript`.

`createdDirID`
The directory ID of the directory that was created.

DESCRIPTION

The `FSpDirCreate` function creates a new directory and returns the directory ID of the new directory in the `createdDirID` parameter. Then `FSpDirCreate` sets the date and time of creation and last modification to the current date and time.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `FSpDirCreate` are

Trap macro	Selector
`_HighLevelHFSDispatch`	$0005

RESULT CODES

`noErr`	0	No error
`dirFulErr`	–33	File directory full
`dskFulErr`	–34	Disk is full
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	Directory not found or incomplete pathname
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`dupFNErr`	–48	Duplicate filename and version
`dirNFErr`	–120	Directory not found or incomplete pathname
`wrgVolTypErr`	–123	Not an HFS volume
`afpAccessDenied`	–5000	User does not have the correct access

FSpDelete

You can use the `FSpDelete` function to delete files and directories.

```
FUNCTION FSpDelete (spec: FSSpec): OSErr;
```

`spec` An `FSSpec` record specifying the file or directory to delete.

DESCRIPTION

The `FSpDelete` function removes a file or directory. If the specified target is a file, both forks of the file are deleted. The file ID reference, if any, is removed.

A file must be closed before you can delete it. Similarly, a directory must be empty before you can delete it. If you attempt to delete an open file or a nonempty directory, `FSpDelete` returns the result code `fBsyErr`. `FSpDelete` also returns the result code `fBsyErr` if the directory has an open working directory associated with it.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `FSpDelete` are

Trap macro	Selector
`_HighLevelHFSDispatch`	$0006

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	File not found
`wPrErr`	–44	Hardware volume lock
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Software volume lock
`fBsyErr`	–47	File busy, directory not empty, or working directory control block open
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access

Accessing Information About Files and Directories

You can use several File Manager routines that accept `FSSpec` records if you want to obtain and set information about files and directories and to manipulate file locking. These routines don't require the file to be open.

FSpGetFInfo

You can use the `FSpGetFInfo` function to obtain the Finder information about a file or directory.

```
FUNCTION FSpGetFInfo (spec: FSSpec; VAR fndrInfo: FInfo): OSErr;
```

`spec` An `FSSpec` record specifying the file or directory whose Finder information is desired.

`fndrInfo` Information used by the Finder.

DESCRIPTION

The `FSpGetFInfo` function returns the Finder information from the volume catalog entry for the specified file or directory. The `FSpGetFInfo` function provides only the original Finder information—the `FInfo` or `DInfo` records, not `FXInfo` or `DXInfo`. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a discussion of Finder information.)

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `FSpGetFInfo` are

Trap macro	Selector
`_HighLevelHFSDispatch`	$0007

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	File not found
`paramErr`	–50	No default volume
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access
`afpObjectTypeErr`	–5025	Directory not found or incomplete pathname

FSpSetFInfo

You can use the `FSpSetFInfo` function to set the Finder information about a file or directory.

```
FUNCTION FSpSetFInfo (spec: FSSpec; fndrInfo: FInfo): OSErr;
```

`spec` An `FSSpec` record specifying the file or directory whose Finder information will be set.

`fndrInfo` Information to be used by the Finder.

DESCRIPTION

The FSpSetFInfo function changes the Finder information in the volume catalog entry for the specified file or directory. FSpSetFInfo allows you to set only the original Finder information—the FInfo or DInfo records, not FXInfo or DXInfo. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a discussion of Finder information.)

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for FSpSetFInfo are

Trap macro	Selector
_HighLevelHFSDispatch	$0008

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access
afpObjectTypeErr	–5025	Object was a directory

FSpSetFLock

You can use the FSpSetFLock function to lock a file.

```
FUNCTION FSpSetFLock (spec: FSSpec): OSErr;
```

spec An FSSpec record specifying the file to lock.

DESCRIPTION

The FSpSetFLock function locks a file. After you lock a file, all new access paths to that file are read-only. This function has no effect on existing access paths.

If the PBHGetVolParms function indicates that the volume supports folder locking (that is, the bHasFolderLock bit of the vMAttrib field is set), you can use FSpSetFLock to lock a directory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `FSpSetFLock` are

Trap macro	Selector
`_HighLevelHFSDispatch`	$0009

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`fnfErr`	–43	File not found
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access to the file
`afpObjectTypeErr`	–5025	Folder locking not supported by volume

FSpRstFLock

You can use the `FSpRstFLock` function to unlock a file.

```
FUNCTION FSpRstFLock (spec: FSSpec): OSErr;
```

`spec` An `FSSpec` record specifying the file to unlock.

DESCRIPTION

The `FSpRstFLock` function unlocks a file.

If the `PBHGetVolParms` function indicates that the volume supports folder locking (that is, the `bHasFolderLock` bit of the `vMAttrib` field is set), you can use `FSpRstFLock` to unlock a directory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `FSpRstFLock` are

Trap macro	Selector
`_HighLevelHFSDispatch`	$000A

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`fnfErr`	–43	File not found
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access to the file
`afpObjectTypeErr`	–5025	Folder locking not supported by volume

FSpRename

You can use the FSpRename function to rename a file or directory.

```
FUNCTION FSpRename (spec: FSSpec; newName: Str255): OSErr;
```

spec	An FSSpec record specifying the file or directory to rename.
newName	The new name of the file or directory.

DESCRIPTION

The FSpRename function changes the name of a file or directory. If a file ID reference for the specified file exists, it remains with the renamed file.

SPECIAL CONSIDERATIONS

If you want to change the name of a new copy of an existing file, you should use the FSpExchangeFiles function instead.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for FSpRename are

Trap macro	Selector
_HighLevelHFSDispatch	$000B

RESULT CODES

noErr	0	No error
dirFulErr	–33	File directory full
dskFulErr	–34	Volume is full
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
dupFNErr	–48	Duplicate filename and version
paramErr	–50	No default volume
fsRnErr	–59	Problem during rename
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file

Moving Files or Directories

The FSpCatMove function allows you to move files and directories within a volume. If the FSSpec routines are not available, you can call the high-level HFS routine CatMove or the low-level HFS routine PBCatMove.

FSpCatMove

You can use the FSpCatMove function to move a file or directory from one location to another on the same volume.

```
FUNCTION FSpCatMove (source: FSSpec; dest: FSSpec): OSErr;
```

source	An FSSpec record specifying the name and location of the file or directory to be moved.
dest	An FSSpec record specifying the name and location of the directory into which the source file or directory is to be moved.

DESCRIPTION

The FSpCatMove function moves the file or directory specified by the source parameter into the directory specified by the dest parameter. The directory ID specified in the parID field of the dest parameter is the directory ID of the parent of the directory into which you want to move the source file or directory. The name field of the dest parameter specifies the name of the directory into which you want to move the source file or directory.

Note
If you don't already know the parent directory ID of the destination directory, it might be easier to use the PBCatMove function, which allows you to specify only the directory ID of the destination directory. ◆

The FSpCatMove function is strictly a file catalog operation; it does not actually change the location of the file or directory on the disk. You cannot use FSpCatMove to move a file or directory to another volume (that is, the vRefNum field in both FSSpec parameters must be the same). Also, you cannot use FSpCatMove to rename files or directories; to rename a file or directory, use FSpRename.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for FSpCatMove are

Trap macro	Selector
_HighLevelHFSDispatch	$000C

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename or attempt to move into a file
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	Target directory is locked
vLckdErr	–46	Software volume lock
dupFNErr	–48	Duplicate filename and version
paramErr	–50	No default volume
badMovErr	–122	Attempt to move into offspring
wrgVolTypErr	–123	Not an HFS volume
afpAccessDenied	–5000	User does not have the correct access to the file

Exchanging the Data in Two Files

The FSpExchangeFiles function allows you to exchange the data in two files.

FSpExchangeFiles

You can use the FSpExchangeFiles function to exchange the data stored in two files on the same volume.

```
FUNCTION FSpExchangeFiles (source: FSSpec; dest: FSSpec): OSErr;
```

source The source file. The contents of this file and its file information are placed in the file specified by the dest parameter.

dest The destination file. The contents of this file and its file information are placed in the file specified by the source parameter.

DESCRIPTION

The FSpExchangeFiles function swaps the data in two files by changing the information in the volume's catalog and, if the files are open, in the file control blocks. You should use FSpExchangeFiles when updating an existing file, so that the file ID remains valid in case the file is being tracked through its file ID. The FSpExchangeFiles function changes the fields in the catalog entries that record the location of the data and the modification dates. It swaps both the data forks and the resource forks.

The FSpExchangeFiles function works on both open and closed files. If either file is open, FSpExchangeFiles updates any file control blocks associated with the file.

Exchanging the contents of two files requires essentially the same access permissions as opening both files for writing.

The files whose data is to be exchanged must both reside on the same volume. If they do not, `FSpExchangeFiles` returns the result code `diffVolErr`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `FSpExchangeFiles` are

Trap macro	Selector
`_HighLevelHFSDispatch`	$000F

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`ioErr`	–36	I/O error
`fnfErr`	–43	File not found
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Volume is locked or read-only
`paramErr`	–50	Function not supported by volume
`volOfflinErr`	–53	Volume is offline
`wrgVolTypErr`	–123	Not an HFS volume
`diffVolErr`	–1303	Files on different volumes
`afpAccessDenied`	–5000	User does not have the correct access
`afpObjectTypeErr`	–5025	Object is a directory, not a file
`afpSameObjectErr`	–5038	Source and destination files are the same

Creating File System Specifications

You can use either the `FSMakeFSSpec` function or the `PBMakeFSSpec` function to create `FSSpec` records. You should always use `FSMakeFSSpec` or `PBMakeFSSpec` to create an `FSSpec` record rather than allocating space and filling out the fields of the record yourself.

FSMakeFSSpec

You can use the `FSMakeFSSpec` function to initialize an `FSSpec` record to particular values for a file or directory.

```
FUNCTION FSMakeFSSpec (vRefNum: Integer; dirID: LongInt;
                       fileName: Str255; VAR spec: FSSpec):
                       OSErr;
```

`vRefNum`	A volume specification. This parameter can contain a volume reference number, a working directory reference number, a drive number, or 0 (to specify the default volume).
`dirID`	A directory specification. This parameter usually specifies the parent directory ID of the target object. If the directory is sufficiently specified by either the `vRefNum` or `fileName` parameter, `dirID` can be set to 0. If you explicitly specify `dirID` (that is, if it has any value other than 0), and if `vRefNum` specifies a working directory reference number, `dirID` overrides the directory ID included in `vRefNum`. If the `fileName` parameter contains an empty string, `FSMakeFSSpec` creates an `FSSpec` record for a directory specified by either the `dirID` or `vRefNum` parameter.
`fileName`	A full or partial pathname. If `fileName` specifies a full pathname, `FSMakeFSSpec` ignores both the `vRefNum` and `dirID` parameters. A partial pathname might identify only the final target, or it might include one or more parent directory names. If `fileName` specifies a partial pathname, then `vRefNum`, `dirID`, or both must be valid.
`spec`	A file system specification to be filled in by `FSMakeFSSpec`.

DESCRIPTION

The `FSMakeFSSpec` function fills in the fields of the `spec` parameter using the information contained in the other three parameters. Call `FSMakeFSSpec` whenever you want to create an `FSSpec` record.

You can pass the input to `FSMakeFSSpec` in any of the ways described in "HFS Specifications" on page 2-28. See Table 2-10 on page 2-36 for information about the way `FSMakeFSSpec` interprets its input.

If the specified volume is mounted and the specified parent directory exists, but the target file or directory doesn't exist in that location, `FSMakeFSSpec` fills in the record and then returns `fnfErr` instead of `noErr`. The record is valid, but it describes a target that doesn't exist. You can use the record for other operations, such as creating a file with the `FSpCreate` function.

In addition to the result codes that follow, `FSMakeFSSpec` can return a number of other File Manager error codes. If your application receives any result code other than `noErr` or `fnfErr`, all fields of the resulting `FSSpec` record are set to 0.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `FSMakeFSSpec` are

Trap macro	Selector
`_HighLevelHFSDispatch`	$0001

RESULT CODES

noErr	0	No error
nsvErr	–35	Volume doesn't exist
fnfErr	–43	File or directory does not exist (FSSpec is still valid)

PBMakeFSSpec

You can use the low-level PBMakeFSSpec function to create an FSSpec record for a file or directory.

```
FUNCTION PBMakeFSSpec (paramBlock: HParmBlkPtr; async: Boolean):
                       OSErr;
```

paramBlock A pointer to a basic HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a file or directory name.
→	ioVRefNum	Integer	A volume specification.
→	ioMisc	LongInt	A pointer to an FSSpec record.
→	ioDirID	LongInt	A parent directory ID.

DESCRIPTION

Given a complete specification for a file or directory, the PBMakeFSSpec function fills in an FSSpec record that identifies the file or directory. (See Table 2-10 on page 2-36 for a detailed description of valid file specifications.)

If the specified volume is mounted and the specified parent directory exists, but the target file or directory doesn't exist in that location, PBMakeFSSpec fills in the record and returns fnfErr instead of noErr. The record is valid, but it describes a target that doesn't exist. You can use the record for another operation, such as creating a file.

In addition to the result codes that follow, PBMakeFSSpec can return a number of different File Manager error codes. When PBMakeFSSpec returns any result other than noErr or fnfErr, all fields of the resulting FSSpec record are set to 0.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBMakeFSSpec are

Trap macro	Selector
_HFSDispatch	$001B

RESULT CODES

noErr	0	No error
nsvErr	–35	Volume doesn't exist
fnfErr	–43	File or directory does not exist (FSSpec is still valid)

High-Level HFS Routines

The File Manager provides a set of high-level file and directory manipulation routines that are available in all operating environments. You may need to use these routines if the FSSpec routines are not available. You do not need to call the Gestalt function to determine if these routines are available.

Each of the high-level HFS routines allows you to specify a file or directory by providing three parameters: a volume specification, a directory specification, and a filename. See "HFS Specifications" on page 2-28 for a complete description of the many ways in which you can set these parameters to pick out a file or directory.

Opening Files

You can use the functions HOpenDF, HOpenRF, and HOpen to open files.

HOpenDF

You can use the HOpenDF function to open the data fork of a file.

```
FUNCTION HOpenDF (vRefNum: Integer; dirID: LongInt;
                  fileName: Str255; permission: SignedByte;
                  VAR refNum: Integer): OSErr;
```

vRefNum	A volume reference number, a working directory reference number, or 0 for the default volume.
dirID	A directory ID.
fileName	The name of the file.
permission	The access mode under which to open the file.
refNum	The file reference number of the opened file.

DESCRIPTION

The HOpenDF function creates an access path to the data fork of a file and returns, in the refNum parameter, an access path reference number to that fork. You can pass that reference number as a parameter to any of the high-level file access routines.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
tmfoErr	–42	Too many files open
fnfErr	–43	File not found
opWrErr	–49	File already open for writing
permErr	–54	Attempt to open locked file for writing
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file

HOpenRF

You can use the HOpenRF function to open the resource fork of file.

```
FUNCTION HOpenRF (vRefNum: Integer; dirID: LongInt;
                  fileName: Str255; permission: SignedByte;
                  VAR refNum: Integer): OSErr;
```

vRefNum	A volume reference number, a working directory reference number, or 0 for the default volume.
dirID	A directory ID.
fileName	The name of the file.
permission	The access mode under which to open the file.
refNum	The file reference number of the opened file.

DESCRIPTION

The HOpenRF function creates an access path to the resource fork of a file. A file reference number for that file is returned in the refNum parameter.

SPECIAL CONSIDERATIONS

Generally, your application should use Resource Manager routines rather than File Manager routines to access a file's resource fork. The HOpenRF function does not read the resource map into memory and is generally useful only for applications (such as utilities that copy files) that need block-level access to a resource fork. In particular, you should not use the resource fork of a file to hold nonresource data. Many parts of the system software assume that a resource fork always contains resource data.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
tmfoErr	–42	Too many files open
fnfErr	–43	File not found
opWrErr	–49	File already open for writing
permErr	–54	Attempt to open locked file for writing
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file

HOpen

You can use the HOpen function to open the data fork of a file. Because HOpen also opens devices, it's safer to use the HOpenDF function instead.

```
FUNCTION HOpen (vRefNum: Integer; dirID: LongInt;
                fileName: Str255; permission: SignedByte;
                VAR refNum: Integer): OSErr;
```

vRefNum	A volume reference number, a working directory reference number, or 0 for the default volume.
dirID	A directory ID.
fileName	The name of the file.
permission	The access mode under which to open the file.
refNum	The file reference number of the opened file.

DESCRIPTION

The HOpen function creates an access path to the data fork of the specified file. A file reference number for that file is returned in the refNum parameter.

▲ **WARNING**
If you use HOpen to try to open a file whose name begins with a period, you might mistakenly open a driver instead; subsequent attempts to write data might corrupt data on the target device. To avoid these problems, you should always use HOpenDF instead of HOpen. ▲

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
tmfoErr	–42	Too many files open
fnfErr	–43	File not found
opWrErr	–49	File already open for writing
permErr	–54	Attempt to open locked file for writing
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file

Creating and Deleting Files and Directories

You can create a file by calling the `HCreate` function and a directory by calling the `DirCreate` function. To delete either a file or a directory, call `HDelete`.

HCreate

You can use the `HCreate` function to create a new file.

```
FUNCTION HCreate (vRefNum: Integer; dirID: LongInt;
                  fileName: Str255; creator: OSType;
                  fileType: OSType): OSErr;
```

`vRefNum`	A volume reference number, a working directory reference number, or 0 for the default volume.
`dirID`	A directory ID.
`fileName`	The name of the new file.
`creator`	The creator of the new file.
`fileType`	The file type of the new file.

DESCRIPTION

The `HCreate` function creates a new file (both forks) with the specified name, creator, and file type. For information on a file's creator and type, see the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.

The new file is unlocked and empty. The date and time of its creation and last modification are set to the current date and time.

Files created using `HCreate` are not automatically opened. If you want to write data to the new file, you must first open the file using a file access routine.

Note
The resource fork of the new file exists but is empty. You'll need to call one of the Resource Manager procedures `CreateResFile`, `HCreateResFile`, or `FSpCreateResFile` to create a resource map in the file before you can open it (by calling one of the Resource Manager functions `OpenResFile`, `HOpenResFile`, or `FSpOpenResFile`). ◆

You should not allow users to give files names that begin with a period (.). This ensures that files can be successfully opened by applications calling `HOpen` instead of `HOpenDF`.

RESULT CODES

`noErr`	0	No error
`dirFulErr`	–33	File directory full
`dskFulErr`	–34	Disk is full
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	Directory not found or incomplete pathname
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`dupFNErr`	–48	Duplicate filename and version
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access
`afpObjectTypeErr`	–5025	A directory exists with that name

DirCreate

You can use the `DirCreate` function to create a new directory.

```
FUNCTION DirCreate (vRefNum: Integer; parentDirID: LongInt;
                    directoryName: Str255;
                    VAR createdDirID: LongInt): OSErr;
```

`vRefNum`	A volume reference number, a working directory reference number, or 0 for the default volume.
`parentDirID`	The directory ID of the parent directory; if it's 0, the new directory is placed in the root directory of the specified volume.
`directoryName`	The name of the new directory.
`createdDirID`	The directory ID of the created directory.

DESCRIPTION

The `DirCreate` function creates a new directory and returns the directory ID of the new directory in the `createdDirID` parameter. The date and time of its creation and last modification are set to the current date and time.

Note

A directory ID, unlike a volume reference number or a working directory reference number, is a `LongInt` value. ◆

RESULT CODES

`noErr`	0	No error
`dirFulErr`	–33	File directory full
`dskFulErr`	–34	Disk is full
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	Directory not found or incomplete pathname
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`dupFNErr`	–48	Duplicate filename and version
`dirNFErr`	–120	Directory not found or incomplete pathname
`wrgVolTypErr`	–123	Not an HFS volume
`afpAccessDenied`	–5000	User does not have the correct access

HDelete

You can use the `HDelete` function to delete a file or directory.

```
FUNCTION HDelete (vRefNum: Integer; dirID: LongInt;
                  fileName: Str255): OSErr;
```

`vRefNum` A volume specification (a volume reference number, a working directory reference number, or 0 for the default volume).

`dirID` The directory ID of the parent of the file or directory to delete.

`fileName` The name of the file or directory to delete.

DESCRIPTION

The `HDelete` function removes a file or directory. If the specified target is a file, both forks of the file are deleted. In addition, if a file ID reference for the specified file exists, that reference is removed.

A file must be closed before you can delete it. Similarly, you cannot delete a directory unless it's empty. If you attempt to delete an open file or a nonempty directory, `HDelete` returns the result code `fBsyErr`. `HDelete` also returns the result code `fBsyErr` if the directory has an open working directory associated with it.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
fBsyErr	–47	File busy, directory not empty, or working directory control block open
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access

Accessing Information About Files and Directories

The File Manager provides a number of high-level HFS routines that allow you to obtain and set information about files and directories and to manipulate file locking. All of the routines described in this section operate on both forks of a file and don't require the file to be open.

HGetFInfo

You can use the `HGetFInfo` function to obtain the Finder information for a file.

```
FUNCTION HGetFInfo (vRefNum: Integer; dirID: LongInt;
                    fileName: Str255; VAR fndrInfo: FInfo):
                    OSErr;
```

`vRefNum` A volume reference number, a working directory reference number, or 0 for the default volume.

`dirID` A directory ID.

`fileName` The name of the file.

`fndrInfo` Information used by the Finder.

DESCRIPTION

The `HGetFInfo` function returns the Finder information stored in the volume's catalog for a file. The `HGetFInfo` function returns only the original Finder information—the `FInfo` record, not `FXInfo`. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a discussion of Finder information.)

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
paramErr	–50	No default volume
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access
afpObjectTypeErr	–5025	Directory not found or incomplete pathname

HSetFInfo

You can use the `HSetFInfo` function to set the Finder information for a file.

```
FUNCTION HSetFInfo (vRefNum: Integer; dirID: LongInt;
                    fileName: Str255; fndrInfo: FInfo): OSErr;
```

`vRefNum` A volume reference number, a working directory reference number, or 0 for the default volume.

`dirID` A directory ID.

`fileName` The name of the file.

`fndrInfo` Information used by the Finder.

DESCRIPTION

The `HSetFInfo` function changes the Finder information stored in the volume's catalog for a file. `HSetFInfo` changes only the original Finder information—the `FInfo` record, not `FXInfo`. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a discussion of Finder information.)

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access
afpObjectTypeErr	–5025	Object was a directory

HSetFLock

You can use the `HSetFLock` function to lock a file.

```
FUNCTION HSetFLock (vRefNum: Integer; dirID: LongInt;
                    fileName: Str255): OSErr;
```

`vRefNum` A volume reference number, a working directory reference number, or 0 for the default volume.

`dirID` A directory ID.

`fileName` The name of the file.

DESCRIPTION

The `HSetFLock` function locks a file. After you lock a file, all new access paths to that file are read-only. This function has no effect on existing access paths.

If the `PBHGetVolParms` function indicates that the volume supports folder locking (that is, the `bHasFolderLock` bit of the `vMAttrib` field is set), you can use `HSetFLock` to lock a directory.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`fnfErr`	–43	File not found
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access to the file
`afpObjectTypeErr`	–5025	Folder locking not supported by volume

HRstFLock

You can use the `HRstFLock` function to unlock a file.

```
FUNCTION HRstFLock (vRefNum: Integer; dirID: LongInt;
                    fileName: Str255): OSErr;
```

`vRefNum` A volume reference number, a working directory reference number, or 0 for the default volume.

`dirID` A directory ID.

`fileName` The name of the file.

DESCRIPTION

The HRstFLock function unlocks a file.

If the PBHGetVolParms function indicates that the volume supports folder locking (that is, the bHasFolderLock bit of the vMAttrib field is set), you can use HRstFLock to unlock a directory.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
vLckdErr	–46	Software volume lock
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file
afpObjectTypeErr	–5025	Folder locking not supported by volume

HRename

You can use the HRename function to rename a file, directory, or volume.

```
FUNCTION HRename (vRefNum: Integer; dirID: LongInt;
                  oldName: Str255; newName: Str255): OSErr;
```

vRefNum	A volume reference number, a working directory reference number, or 0 for the default volume.
dirID	A directory ID.
oldName	An existing filename, directory name, or volume name.
newName	The new filename, directory name, or volume name.

DESCRIPTION

The HRename function changes the name of a file, directory, or volume. Given the name of a file or directory in oldName, HRename changes it to the name in newName. Given a volume name or a volume reference number, it changes the name of the volume to the name in newName. Access paths currently in use aren't affected.

SPECIAL CONSIDERATIONS

You cannot use HRename to change the directory in which a file resides. If you're renaming a volume, make sure that both names end with a colon.

Note
If a file ID reference exists for a file you are renaming, the file ID remains with the renamed file. ◆

RESULT CODES

noErr	0	No error
dirFulErr	–33	File directory full
dskFulErr	–34	Volume is full
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
dupFNErr	–48	Duplicate filename
paramErr	–50	No default volume
fsRnErr	–59	Problem during rename
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file

Moving Files or Directories

The high-level HFS function `CatMove` allows you to move files and directories within a volume.

CatMove

You can use the `CatMove` function to move files or directories from one directory to another on the same volume.

```
FUNCTION CatMove (vRefNum: Integer; dirID: LongInt;
                  oldName: Str255; newDirID: LongInt;
                  newName: Str255): OSErr;
```

`vRefNum` A volume reference number, a working directory reference number, or 0 for the default volume.

`dirID` A directory ID.

`oldName` An existing filename or directory name.

`newDirID` If `newName` is empty, the directory ID of the target directory; otherwise, the parent directory ID of the target directory.

`newName` The name of the directory to which the file or directory is to be moved.

DESCRIPTION

The `CatMove` function moves a file or directory from one directory to another within a volume. `CatMove` is strictly a file catalog operation; it does not actually change the location of the file or directory on the disk.

The newName parameter specifies the name of the directory to which the file or directory is to be moved. If a valid directory name is provided for newName, the destination directory's parent directory is specified in newDirID. However, you can specify an empty name for newName, in which case newDirID should be set to the directory ID of the destination directory.

Note

It is usually simplest to specify the destination directory by passing its directory ID in the newDirID parameter and by setting newName to an empty name. To specify an empty name, set newName to ':'. ◆

The CatMove function cannot move a file or directory to another volume (that is, the vRefNum parameter is used in specifying both the source and the destination). Also, you cannot use it to rename files or directories; to rename a file or directory, use HRename.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename or attempt to move into a file
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	Target directory is locked
vLckdErr	–46	Software volume lock
dupFNErr	–48	Duplicate filename and version
paramErr	–50	No default volume
badMovErr	–122	Attempt to move into offspring
wrgVolTypErr	–123	Not an HFS volume
afpAccessDenied	–5000	User does not have the correct access to the file

Maintaining Working Directories

The File Manager provides several functions that allow you to manipulate working directories. Working directories are used internally by the File Manager; in general, your application should not create or directly access working directories. For more information about working directories, see "Working Directory Reference Numbers," beginning on page 2-26.

OpenWD

You can use the OpenWD function to create a working directory.

```
FUNCTION OpenWD (vRefNum: Integer; dirID: LongInt;
                 procID: LongInt; VAR wdRefNum: Integer): OSErr;
```

vRefNum	A volume reference number, a working directory reference number, or 0 for the default volume.
dirID	A directory ID.
procID	A working directory user identifier. You should use your application's signature as the user identifier.
wdRefNum	On exit, the working directory reference number.

DESCRIPTION

The OpenWD function creates a working directory that corresponds to the specified directory. It returns in wdRefNum a working directory reference number that can be used in subsequent File Manager calls.

If a working directory having the specified user identifier already exists for the specified directory, no new working directory is opened; instead, the existing working directory reference number is returned in wdRefNum. If the specified directory already has a working directory with a different user identifier, a new working directory reference number is returned.

If the directory specified by the dirID parameter is the volume's root directory, no working directory is created; instead, the volume reference number is returned in the wdRefNum parameter.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
fnfErr	–43	No such directory
tmwdoErr	–121	Too many working directories open
afpAccessDenied	–5000	User does not have the correct access to the file

CloseWD

You can use the CloseWD function to close a working directory.

```
FUNCTION CloseWD (wdRefNum: Integer): OSErr;
```

wdRefNum	A working directory reference number.

DESCRIPTION

The CloseWD function releases the specified working directory.

Note

If you specify a volume reference number in the wdRefNum parameter, CloseWD does nothing. ◆

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
rfNumErr	–51	Bad working directory reference number

GetWDInfo

You can use the GetWDInfo function to get information about a working directory.

```
FUNCTION GetWDInfo (wdRefNum: Integer; VAR vRefNum: Integer;
                    VAR dirID: LongInt; VAR procID: LongInt):
                    OSErr;
```

wdRefNum	A working directory reference number.
vRefNum	If nonzero on input, a volume reference number or drive number. On output, the volume reference number of the working directory.
dirID	On output, the directory ID of the specified working directory.
procID	The working directory user identifier.

DESCRIPTION

The GetWDInfo function returns information about the specified working directory. You can use GetWDInfo to convert a working directory reference number to its corresponding volume reference number and directory ID.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
rfNumErr	–51	Bad working directory reference number

Low-Level HFS Routines

The File Manager provides a set of low-level file and directory manipulation routines that are available in all operating environments. You do not need to call the Gestalt function to determine if these routines are available.

These routines exchange parameters with your application through a parameter block. When you call a low-level routine, you pass the address of the appropriate parameter block to the routine.

Some low-level HFS routines can run either asynchronously or synchronously. Each of these routines comes in three versions: one version requires the async parameter, and two have the suffix Async or Sync added to their names. For more information about the differences between the three versions, see “Low-Level File Access Routines” on

page 2-121. Only the first version of these routines is documented in this section. See "Summary of the File Manager," beginning on page 2-243, for a listing that includes all three versions.

Assembly-Language Note

See the assembly-language note on page 2-121 for details on calling these routines from assembly language. ◆

Opening Files

You can use the functions PBHOpenDF, PBHOpenRF, and PBHOpen to open files.

PBHOpenDF

You can use the PBHOpenDF function to open the data fork of a file.

```
FUNCTION PBHOpenDF (paramBlock: HParmBlkPtr; async: Boolean):
                    OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
←	ioRefNum	Integer	A file reference number.
→	ioPermssn	SignedByte	The read / write permission.
→	ioDirID	LongInt	A parent directory ID.

DESCRIPTION

The PBHOpenDF function creates an access path to the data fork of a file and returns a file reference number in the ioRefNum field. PBHOpenDF is exactly like the PBHOpen function except that PBHOpenDF allows you to open a file whose name begins with a period (.).

You can open a path for writing even if it accesses a file on a locked volume, and no error is returned until a PBWrite, PBSetEOF, or PBAllocate call is made.

If you attempt to open a locked file for writing, PBHOpenDF returns the result code permErr. If you request exclusive read / write permission but another access path is already open, PBHOpenDF returns the reference number of the existing access path in ioRefNum and opWrErr as its function result. You should not use this reference number unless your application originally opened the file.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBHOpenDF are

Trap macro	Selector
_HFSDispatch	$001A

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
tmfoErr	–42	Too many files open
fnfErr	–43	File not found
opWrErr	–49	File already open for writing
permErr	–54	Attempt to open locked file for writing
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file

PBHOpenRF

You can use the PBHOpenRF function to open the resource fork of file.

```
FUNCTION PBHOpenRF (paramBlock: HParmBlkPtr; async: Boolean):
                    OSErr;
```

paramBlock A pointer to a basic HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
←	ioRefNum	Integer	A file reference number.
→	ioPermssn	SignedByte	The read/write permission.
→	ioDirID	LongInt	A directory ID.

DESCRIPTION

The PBHOpenRF function creates an access path to the resource fork of a file and returns a file reference number in the ioRefNum field.

SPECIAL CONSIDERATIONS

Generally your application should use Resource Manager routines rather than File Manager routines to access a file's resource fork. The PBHOpenRF function does not read the resource map into memory and is generally useful only for applications (such as utilities that copy files) that need block-level access to a resource fork. In particular, you should not use the resource fork of a file to hold nonresource data. Many parts of the system software assume that a resource fork always contains resource data.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBHOpenRF is _HOpenRF.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
tmfoErr	–42	Too many files open
fnfErr	–43	File not found
opWrErr	–49	File already open for writing
permErr	–54	Attempt to open locked file for writing
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file

PBHOpen

You can use the PBHOpen function to open the data fork of a file. Because PBHOpen will also open devices, it's safer to use the PBHOpenDF function instead.

```
FUNCTION PBHOpen (paramBlock: HParmBlkPtr; async: Boolean): OSErr;
```

paramBlock A pointer to a basic HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
←	ioRefNum	Integer	A file reference number.
→	ioPermssn	SignedByte	The read/write permission.
→	ioDirID	LongInt	A directory ID.

DESCRIPTION

The PBHOpen function creates an access path to the data fork of the specified file and returns a file reference number in the ioRefNum field.

You can open a path for writing even if it accesses a file on a locked volume, and no error is returned until a PBWrite, PBSetEOF, or PBAllocate call is made.

If you attempt to open a locked file for writing, PBHOpen returns the result code permErr. If you request exclusive read/write permission but another access path is already open, PBHOpen returns the reference number of the existing access path in ioRefNum and opWrErr as its function result. You should not use this reference number unless your application originally opened the file.

▲ **WARNING**
If you use PBHOpen to try to open a file whose name begins with a period, you might mistakenly open a driver instead; subsequent attempts to write data might corrupt data on the target device. To avoid these problems, you should always use PBHOpenDF instead of PBHOpen. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBHOpen is _HOpen.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
tmfoErr	–42	Too many files open
fnfErr	–43	File not found
opWrErr	–49	File already open for writing
permErr	–54	Attempt to open locked file for writing
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file

Creating and Deleting Files and Directories

You can create a file by calling the PBHCreate function and a directory by calling the PBDirCreate function. To delete either a file or a directory, use PBHDelete.

PBHCreate

You can use the PBHCreate function to create a new file.

```
FUNCTION PBHCreate (paramBlock: HParmBlkPtr; async: Boolean):
                    OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioDirID`	`LongInt`	A directory ID.

DESCRIPTION

The `PBHCreate` function creates a new file (both forks); the new file is unlocked and empty. The date and time of its creation and last modification are set to the current date and time. If the file created isn't temporary (that is, if it will exist after the user quits the application), the application should call `PBHSetFInfo` (after `PBHCreate`) to fill in the information needed by the Finder.

Files created using `PBHCreate` are not automatically opened. If you want to write data to the new file, you must first open the file using a file access routine (such as `PBHOpenDF`).

Note

The resource fork of the new file exists but is empty. You'll need to call one of the Resource Manager procedures `CreateResFile`, `HCreateResFile`, or `FSpCreateResFile` to create a resource map in the file before you can open it (by calling one of the Resource Manager functions `OpenResFile`, `HOpenResFile`, or `FSpOpenResFile`). ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBHCreate` is `_HCreate`.

RESULT CODES

`noErr`	0	No error
`dirFulErr`	–33	File directory full
`dskFulErr`	–34	Disk is full
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	Directory not found or incomplete pathname
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`dupFNErr`	–48	Duplicate filename and version
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access
`afpObjectTypeErr`	–5025	A directory exists with that name

PBDirCreate

You can use the `PBDirCreate` function to create a new directory.

```
FUNCTION PBDirCreate (paramBlock: HParmBlkPtr; async: Boolean):
                        OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
↔	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
↔	`ioDirID`	`LongInt`	A directory ID.

DESCRIPTION

The `PBDirCreate` function is identical to `PBHCreate` except that it creates a new directory instead of a file. You can specify the parent of the directory to be created in `ioDirID`; if it's 0, the new directory is placed in the root directory of the specified volume. The directory ID of the new directory is returned in `ioDirID`. The date and time of its creation and last modification are set to the current date and time.

Note
A directory ID, unlike a volume reference number or a working directory reference number, is a `LongInt` value. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBDirCreate` are

Trap macro	Selector
`_HFSDispatch`	$0006

RESULT CODES

`noErr`	0	No error
`dirFulErr`	–33	File directory full
`dskFulErr`	–34	Disk is full
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	Directory not found or incomplete pathname
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`dupFNErr`	–48	Duplicate filename and version
`dirNFErr`	–120	Directory not found or incomplete pathname
`wrgVolTypErr`	–123	Not an HFS volume
`afpAccessDenied`	–5000	User does not have the correct access

PBHDelete

You can use the `PBHDelete` function to delete a file or directory.

```
FUNCTION PBHDelete (paramBlock: HParmBlkPtr; async: Boolean):
                    OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioDirID`	`LongInt`	A directory ID.

DESCRIPTION

The `PBHDelete` function removes a file or directory. If the specified target is a file, both forks of the file are deleted. In addition, if a file ID reference for the specified file exists, that file ID reference is also removed.

A file must be closed before you can delete it. Similarly, you cannot delete a directory unless it's empty. If you attempt to delete an open file or a nonempty directory, `PBHDelete` returns the result code `fBsyErr`. `PBHDelete` also returns `fBsyErr` if you attempt to delete a directory that has an open working directory associated with it.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBHDelete` is `_HDelete`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	File not found
`wPrErr`	–44	Hardware volume lock
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Software volume lock
`fBsyErr`	–47	File busy, directory not empty, or working directory control block open
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access

Accessing Information About Files and Directories

The File Manager provides a number of low-level HFS routines that allow you to obtain and set information about files and directories and to manipulate file locking. All of the routines described in this section operate on both forks of a file and don't require the file to be open.

PBGetCatInfo

You can use the `PBGetCatInfo` function to get information about the files and directories in a file catalog.

```
FUNCTION PBGetCatInfo (paramBlock: CInfoPBPtr; async: Boolean):
                        OSErr;
```

`paramBlock` A pointer to a catalog information parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block for files

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
↔	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
←	`ioFRefNum`	`Integer`	A file reference number.
→	`ioFDirIndex`	`Integer`	An index.
←	`ioFlAttrib`	`SignedByte`	The file attributes.
←	`ioFlFndrInfo`	`FInfo`	Information used by the Finder.
↔	`ioDirID`	`LongInt`	On input, a directory ID. On output, a file ID.
←	`ioFlStBlk`	`Integer`	The first allocation block of the data fork.
←	`ioFlLgLen`	`LongInt`	The logical end-of-file of the data fork.
←	`ioFlPyLen`	`LongInt`	The physical end-of-file of the data fork.
←	`ioFlRStBlk`	`Integer`	The first allocation block of the resource fork.
←	`ioFlRLgLen`	`LongInt`	The logical end-of-file of the resource fork.
←	`ioFlRPyLen`	`LongInt`	The physical end-of-file of the resource fork.
←	`ioFlCrDat`	`LongInt`	The date and time of creation.
←	`ioFlMdDat`	`LongInt`	The date and time of the last modification.
←	`ioFlBkDat`	`LongInt`	The date and time of the last backup.
←	`ioFlXFndrInfo`	`FXInfo`	Additional information used by the Finder.
←	`ioFlParID`	`LongInt`	The directory ID of the parent directory.
←	`ioFlClpSiz`	`LongInt`	The file's clump size.

Parameter block for directories

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
↔	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioFDirIndex	Integer	An index.
←	ioFlAttrib	SignedByte	The directory attributes.
←	ioACUser	SignedByte	The directory access rights.
←	ioDrUsrWds	DInfo	Information used by the Finder.
↔	ioDrDirID	LongInt	The directory ID.
←	ioDrNmFls	Integer	The number of files in the directory.
←	ioDrCrDat	LongInt	The date and time of creation.
←	ioDrMdDat	LongInt	The date and time of the last modification.
←	ioDrBkDat	LongInt	The date and time of the last backup.
←	ioDrFndrInfo	DXInfo	Additional information used by the Finder.
←	ioDrParID	LongInt	The directory ID of the parent directory.

DESCRIPTION

The PBGetCatInfo function returns information about a file or directory, depending on the values you specify in the ioFDirIndex, ioNamePtr, ioVRefNum, and ioDirID or ioDrDirID fields. If you need to determine whether the information returned is for a file or a directory, you can test bit 4 of the ioFlAttrib field; if that bit is set, the information returned describes a directory.

The PBGetCatInfo function selects a file or directory according to these rules:

- If the value of ioFDirIndex is positive, PBGetCatInfo returns information about the file or directory whose directory index is ioFDirIndex in the directory specified by ioVRefNum (this will be the root directory if a volume reference number is provided).
- If the value of ioFDirIndex is 0, PBGetCatInfo returns information about the file or directory specified by ioNamePtr in the directory specified by ioVRefNum (again, this will be the root directory if a volume reference number is provided).
- If the value of ioFDirIndex is negative, PBGetCatInfo ignores ioNamePtr and returns information about the directory specified by ioDrDirID.

With files, PBGetCatInfo is similar to PBHGetFInfo but returns some additional information. If the file is open, the reference number of the first access path found is returned in ioFRefNum, and the name of the file is returned in ioNamePtr (unless ioNamePtr is NIL). The file's attributes are returned in the ioFlAttrib field. See the description of the fields of the CInfoPBRec data type (beginning on page 2-101) for the meaning of the bits in this field.

Note

When you get information about a file, the ioDirID field contains the file ID on exit from PBGetCatInfo. You might need to save the value of ioDirID before calling PBGetCatInfo if you make subsequent calls with the same parameter block. ◆

With directories, PBGetCatInfo returns information such as the directory attributes and, for server volumes, the directory access privileges of the user. The directory attributes are encoded by bits in the ioFlAttrib field and have these meanings:

Bit	Meaning
0	Set if the directory is locked
1	Reserved
2	Set if the directory is within a shared area of the directory hierarchy
3	Set if the directory is a share point that is mounted by some user
4	Set if the item is a directory
5	Set if the directory is a share point
6–7	Reserved

Note

These bits in the ioFlAttrib field for directories are read-only. You cannot alter directory attributes by setting these bits using PBSetCatInfo. Instead, you can call PBHSetFLock and PBHRstFLock to lock and unlock a directory, and PBShare and PBUnshare to enable and disable file sharing on local directories. ◆

The PBGetCatInfo function returns the directory access rights in the ioACUser field only for shared volumes. As a result, you should set this field to 0 before calling PBGetCatInfo.

You can also use PBGetCatInfo to determine whether a file has a file ID reference. The value of the file ID is returned in the ioDirID field. Because that parameter could also represent a directory ID, call PBResolveFileIDRef to see if the value is a real file ID. If you want to determine whether a file ID reference exists for a file and create one if it doesn't, use PBCreateFileIDRef, which will either create a file ID or return fidExists.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBGetCatInfo are

Trap macro	Selector
_HFSDispatch	$0009

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
paramErr	–50	No default volume
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access
afpObjectTypeErr	–5025	Directory not found or incomplete pathname

PBSetCatInfo

You can use the `PBSetCatInfo` function to modify information about files and directories.

```
FUNCTION PBSetCatInfo (paramBlock: CInfoPBPtr; async: Boolean):
                        OSErr;
```

`paramBlock` A pointer to a catalog information parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block for files

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioFlFndrInfo`	`FInfo`	Information used by the Finder.
→	`ioDirID`	`LongInt`	The directory ID.
→	`ioFlCrDat`	`LongInt`	The date and time of creation.
→	`ioFlMdDat`	`LongInt`	The date and time of the last modification.
→	`ioFlBkDat`	`LongInt`	The date and time of the last backup.
→	`ioFlXFndrInfo`	`FXInfo`	Additional information used by the Finder.

Parameter block for directories

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioDrUsrWds`	`DInfo`	Information used by the Finder.
→	`ioDrDirID`	`LongInt`	The directory ID.
→	`ioDrCrDat`	`LongInt`	The date and time of creation.
→	`ioDrMdDat`	`LongInt`	The date and time of the last modification.
→	`ioDrBkDat`	`LongInt`	The date and time of the last backup.
→	`ioDrFndrInfo`	`DXInfo`	Additional information used by the Finder.

DESCRIPTION

The `PBSetCatInfo` function sets information about a file or directory. When used to set information about a file, it works much as `PBHSetFInfo` does, but lets you set some additional information.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBSetCatInfo` are

Trap macro	Selector
`_HFSDispatch`	$000A

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	File not found
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Volume is locked or read-only
`paramErr`	–50	No default volume
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access

PBHGetFInfo

You can use the `PBHGetFInfo` function to obtain information about a file.

```
FUNCTION PBHGetFInfo (paramBlock: HParmBlkPtr; async: Boolean):
                        OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
↔	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
←	`ioFRefNum`	`Integer`	A file reference number.
→	`ioFDirIndex`	`Integer`	An index.
←	`ioFlAttrib`	`SignedByte`	The file attributes.
←	`ioFlFndrInfo`	`FInfo`	Information used by the Finder.
↔	`ioDirID`	`LongInt`	On input, a directory ID; on output, a file ID.
←	`ioFlStBlk`	`Integer`	The first allocation block of the data fork.
←	`ioFlLgLen`	`LongInt`	The logical end-of-file of the data fork.
←	`ioFlPyLen`	`LongInt`	The physical end-of-file of the data fork.
←	`ioFlRStBlk`	`Integer`	The first allocation block of the resource fork.
←	`ioFlRLgLen`	`LongInt`	The logical end-of-file of the resource fork.
←	`ioFlRPyLen`	`LongInt`	The physical end-of-file of the resource fork.
←	`ioFlCrDat`	`LongInt`	The date and time of creation.
←	`ioFlMdDat`	`LongInt`	The date and time of last modification.

DESCRIPTION

If the value of `ioFDirIndex` is positive, the `PBHGetFInfo` function returns information about the file whose directory index is `ioFDirIndex` on the volume specified by `ioVRefNum` in the directory specified by `ioDirID`. You should call `PBHGetFInfo` just before `PBHSetFInfo`, so that the current information is present in the parameter block.

Note

If a working directory reference number is specified in `ioVRefNum`, the File Manager returns information about the file whose directory index is `ioFDirIndex` in the specified directory. ◆

If the value of `ioFDirIndex` is negative or 0, the `PBHGetFInfo` function returns information about the file having the name pointed to by `ioNamePtr` on the volume specified by `ioVRefNum`. If the file is open, the reference number of the first access path found is returned in `ioFRefNum`, and the name of the file is returned in `ioNamePtr` (unless `ioNamePtr` is `NIL`).

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBHGetFInfo` is `_HGetFileInfo`.

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`ioErr`	–36	I/O error
`bdNamErr`	–37	Bad filename
`fnfErr`	–43	File not found
`paramErr`	–50	No default volume
`dirNFErr`	–120	Directory not found or incomplete pathname
`afpAccessDenied`	–5000	User does not have the correct access
`afpObjectTypeErr`	–5025	Directory not found or incomplete pathname

PBHSetFInfo

You can use the `PBHSetFInfo` function to set information for a file.

```
FUNCTION PBHSetFInfo (paramBlock: HParmBlkPtr; async: Boolean):
                        OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioFlFndrInfo	FInfo	Information used by the Finder.
→	ioDirID	LongInt	A directory ID.
→	ioFlCrDat	LongInt	The date and time of creation.
→	ioFlMdDat	LongInt	The date and time of last modification.

DESCRIPTION

The PBHSetFInfo function sets information (including the date and time of creation and modification, and information needed by the Finder) about the file having the name pointed to by ioNamePtr on the volume specified by ioVRefNum. You should call PBHGetFInfo just before PBHSetFInfo, so that the current information is present in the parameter block.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBHSetFInfo is _HSetFileInfo.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access
afpObjectTypeErr	–5025	Object was a directory

PBHSetFLock

You can use the PBHSetFLock function to lock a file.

```
FUNCTION PBHSetFLock (paramBlock: HParmBlkPtr; async: Boolean):
                        OSErr;
```

paramBlock A pointer to a basic HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioDirID	LongInt	A directory ID.

DESCRIPTION

The PBHSetFLock function locks the file with the name pointed to by ioNamePtr on the volume specified by ioVRefNum. After you lock a file, all new access paths to that file are read-only. Access paths currently in use aren't affected.

If the PBHGetVolParms function indicates that the volume supports folder locking (that is, the bHasFolderLock bit of the vMAttrib field is set), you can use PBHSetFLock to lock a directory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBHSetFLock is _HSetFLock.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
vLckdErr	–46	Software volume lock
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file
afpObjectTypeErr	–5025	Folder locking not supported by volume

PBHRstFLock

You can use the PBHRstFLock function to unlock a file.

```
FUNCTION PBHRstFLock (paramBlock: HParmBlkPtr; async: Boolean):
                        OSErr;
```

paramBlock A pointer to a basic HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioDirID	LongInt	A directory ID.

DESCRIPTION

The PBHRstFLock function unlocks the file with the name pointed to by ioNamePtr on the volume specified by ioVRefNum. Access paths currently in use aren't affected.

If the PBHGetVolParms function indicates that the volume supports folder locking (that is, the bHasFolderLock bit of the vMAttrib field is set), you can use PBHRstFLock to unlock a directory.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for PBHRstFLock is _HRstFLock.

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
vLckdErr	–46	Software volume lock
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access to the file
afpObjectTypeErr	–5025	Folder locking not supported by volume

PBHRename

You can use the PBHRename function to rename a file, directory, or volume.

```
FUNCTION PBHRename (paramBlock: HParmBlkPtr; async: Boolean):
                    OSErr;
```

paramBlock A pointer to a basic HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioMisc	Ptr	A pointer to the new name for the file.
→	ioDirID	LongInt	A directory ID.

DESCRIPTION

Given a pointer to the name of a file or directory in `ioNamePtr`, `PBHRename` changes it to the name pointed to by `ioMisc`. Given a pointer to a volume name in `ioNamePtr` or a volume reference number in `ioVRefNum`, it changes the name of the volume to the name pointed to by `ioMisc`.

Note

If a file ID reference exists for the file being renamed, the file ID remains with the file. ◆

IMPORTANT

You cannot use `PBHRename` to change the directory in which a file is located. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for `PBHRename` is `_HRename`.

RESULT CODES

noErr	0	No error
dirFulErr	–33	File directory full
dskFulErr	–34	Volume is full
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
dupFNErr	–48	Duplicate filename and version
paramErr	–50	No default volume
fsRnErr	–59	Problem during rename
dirNFErr	–120	Directory not found or incomplete pathname
afpAccessDenied	–5000	User does not have the correct access

Moving Files or Directories

The low-level HFS function `PBCatMove` allows you to move files and directories within a volume.

PBCatMove

You can use the PBCatMove function to move files or directories from one directory to another on the same volume.

```
FUNCTION PBCatMove (paramBlock: CMovePBPtr; async: Boolean):
                    OSErr;
```

paramBlock A pointer to a catalog move parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to the name of the file or directory to be moved.
→	ioVRefNum	Integer	A volume specification.
→	ioNewName	StringPtr	A pointer to the name of the directory into which the file or directory is to be moved.
→	ioNewDirID	LongInt	The directory ID of the directory into which the file or directory is to be moved, if ioNewName is NIL. If ioNewName is not NIL, this is the parent directory ID of the directory into which the file or directory is to be moved.
→	ioDirID	LongInt	The directory ID of the file or directory to be moved.

DESCRIPTION

The PBCatMove function moves a file or directory from one directory to another within a volume. PBCatMove is strictly a file catalog operation; it does not actually change the location of the file or directory on the disk.

The source file or directory should be specified by its volume, parent directory ID, and partial pathname. Pass a volume specification in ioVRefNum. Pass the parent directory ID in the ioDirID field and a pointer to the partial pathname in the ioNamePtr field.

The name of the directory into which the file or directory is to be moved is specified by the ioNewName field. If a valid directory name is provided for ioNewName, the destination directory's parent directory is specified in ioNewDirID. However, you can specify NIL for ioNewName, in which case ioNewDirID should be set to the directory ID of the destination directory itself.

Note

It is usually simplest to specify the destination directory by passing its directory ID in the ioNewDirID field and by setting ioNewName to NIL. ◆

The PBCatMove function cannot move a file or directory to another volume (that is, ioVRefNum is used in specifying both the source and the destination). Also, you cannot use it to rename files or directories; to rename a file or directory, use PBHRename.

If a file ID reference exists for the file, the file ID reference remains with the moved file.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBCatMove are

Trap macro	Selector
_HFSDispatch	$0005

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
ioErr	–36	I/O error
bdNamErr	–37	Bad filename or attempt to move into a file
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	Target directory is locked
vLckdErr	–46	Software volume lock
dupFNErr	–48	Duplicate filename and version
paramErr	–50	No default volume
badMovErr	–122	Attempt to move into offspring
wrgVolTypErr	–123	Not an HFS volume
afpAccessDenied	–5000	User does not have the correct access

Maintaining Working Directories

The File Manager provides several low-level functions that allow you to manipulate working directories. Working directories are used internally by the File Manager; in general, your application should not create or directly access working directories. For more information about working directories, see "Working Directory Reference Numbers," beginning on page 2-26.

PBOpenWD

You can use the PBOpenWD function to create a working directory.

```
FUNCTION PBOpenWD (paramBlock: WDPBPtr; async: Boolean): OSErr;
```

paramBlock A pointer to a working directory parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
↔	ioVRefNum	Integer	A volume specification.
→	ioWDProcID	LongInt	The working directory user identifier.
→	ioWDDirID	LongInt	The working directory's directory ID.

DESCRIPTION

The PBOpenWD function creates a working directory that corresponds to the directory specified by ioVRefNum, ioWDDirID, and ioWDProcID. (You can also specify the directory using a combination of partial pathname and directory ID.) PBOpenWD returns in ioVRefNum a working directory reference number that can be used in subsequent File Manager calls.

If a working directory having the specified user identifier already exists for the specified directory, no new working directory is opened; instead, the existing working directory reference number is returned in ioVRefNum. If the specified directory already has a working directory with a different user identifier, a new working directory reference number is returned.

If the directory specified by the ioWDDirID parameter is the volume's root directory, no working directory is created; instead, the volume reference number is returned in the ioVRefNum parameter.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBOpenWD are

Trap macro	Selector
_HFSDispatch	$0001

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
fnfErr	–43	No such directory
tmwdoErr	–121	Too many working directories open
afpAccessDenied	–5000	User does not have the correct access

PBCloseWD

You can use the PBCloseWD function to close a working directory.

```
FUNCTION PBCloseWD (paramBlock: WDPBPtr; async: Boolean): OSErr;
```

paramBlock A pointer to a working directory parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioVRefNum	Integer	A working directory reference number.

DESCRIPTION

The PBCloseWD function releases the working directory whose working directory reference number is specified in ioVRefNum.

Note

If you specify a volume reference number in the ioVRefNum field, PBCloseWD does nothing. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBCloseWD are

Trap macro	Selector
_HFSDispatch	$0002

RESULT CODES

noErr	0	No error
nsvErr	–35	No such volume
rfNumErr	–51	Bad working directory reference number

PBGetWDInfo

You can use the PBGetWDInfo function to get information about a working directory.

```
FUNCTION PBGetWDInfo (paramBlock: WDPBPtr; async: Boolean): OSErr;
```

paramBlock A pointer to a working directory parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
←	ioNamePtr	StringPtr	A pointer to a pathname.
↔	ioVRefNum	Integer	A volume specification.
→	ioWDIndex	Integer	An index.
↔	ioWDProcID	LongInt	The working directory user identifier.
↔	ioWDVRefNum	Integer	The volume reference number for the working directory.
←	ioWDDirID	LongInt	The working directory's directory ID.

DESCRIPTION

The `PBGetWDInfo` function returns information about the specified working directory. The working directory can be specified either by its working directory reference number in `ioVRefNum` (in which case the value of `ioWDIndex` should be 0), or by its index number in `ioWDIndex`. In the latter case, if the value of `ioVRefNum` is not 0, it's interpreted as a volume specification, and only working directories on that volume are indexed.

The `ioWDVRefNum` field always returns the volume reference number. The `ioVRefNum` field contains a working directory reference number when a working directory reference number is passed in that field; otherwise, it returns a volume reference number. `PBGetWDInfo` returns a pointer to the volume's name in the `ioNamePtr` field. You should pass a pointer to a `Str31` value if you want that name returned. If you pass `NIL` in the `ioNamePtr` field, no volume name is returned.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBGetWDInfo` are

Trap macro	Selector
`_HFSDispatch`	$0007

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	No such volume
`rfNumErr`	–51	Bad working directory reference number

Searching a Catalog

The low-level HFS function `PBCatSearch` allows you to search a volume using a particular set of search criteria.

PBCatSearch

The `PBCatSearch` function searches a volume's catalog file using a set of search criteria that you specify. It builds a list of all files or directories that meet your specifications.

```
FUNCTION PBCatSearch (paramBlock: HParmBlkPtr; async: Boolean):
                        OSErr;
```

`paramBlock` A pointer to a `csParam` variant of an HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a volume name.
→	ioVRefNum	Integer	A volume specification.
→	ioMatchPtr	FSSpecArrayPtr	A pointer to an array of matches.
→	ioReqMatchCount	LongInt	The maximum match count.
←	ioActMatchCount	LongInt	The actual match count.
→	ioSearchBits	LongInt	Enable bits for fields in criteria records.
→	ioSearchInfo1	CInfoPBPtr	The values and lower bounds.
→	ioSearchInfo2	CInfoPBPtr	The masks and upper bounds.
→	ioSearchTime	LongInt	The maximum allowed search time.
↔	ioCatPosition	CatPositionRec	The current catalog position.
→	ioOptBuffer	Ptr	A pointer to optional read buffer.
→	ioOptBufSize	LongInt	The length of optional read buffer.

DESCRIPTION

The PBCatSearch function searches the volume you specify for files or directories that match two coordinated sets of selection criteria. PBCatSearch returns (in the ioMatchPtr field) a pointer to an array of FSSpec records identifying the files and directories that match the criteria.

If the catalog file changes between two timed calls to PBCatSearch (that is, when you are using ioSearchTime and ioCatPosition to search a volume in segments and the catalog file changes between searches), PBCatSearch returns a result code of catChangedErr and no matches. Depending on what has changed on the volume, ioCatPosition might be invalid, most likely by a few entries in one direction or another. You can continue the search, but you risk either skipping some entries or reading some twice.

When PBCatSearch has searched the entire volume, it returns eofErr. If it exits because it either spends the maximum time allowed by ioSearchTime or finds the maximum number of matches allowed by ioReqMatchCount, it returns noErr. You can specify a value of 0 in the ioSearchTime field to indicate that no time limit is to be enforced.

SPECIAL CONSIDERATIONS

Not all volumes support the PBCatSearch function. Before you call PBCatSearch to search a particular volume, you should call the PBHGetVolParms function to determine whether that volume supports PBCatSearch. See page 2-148 for details on calling PBHGetVolParms.

Even though AFP volumes support PBCatSearch, they do not support all of its features that are available on local volumes. These restrictions apply to AFP volumes:

- AFP volumes do not use the ioSearchTime field. Current versions of the AppleShare server software search for 1 second or until 4 matches are found. The AppleShare workstation software keeps requesting the appropriate number of matches until the server returns either the number specified in the ioReqMatchCount field or an error.

- AFP volumes do not support both logical and physical fork lengths. If you request a search using the length of a fork, the actual minimum length used is the smallest of the values in the logical and physical fields of the `ioSearchInfo1` record and the actual maximum length used is the largest of the values in the logical and physical fields of the `ioSearchInfo2` record.
- The `fsSBNegate` bit of the `ioSearchBits` field is ignored during searches of remote volumes that support AFP version 2.1.
- If the AFP server returns `afpCatalogChanged`, the catalog position record returned to your application (in the `ioCatPosition` field) is the same one you passed to `PBCatSearch`. You should clear the `initialize` field of that record to restart the search from the beginning.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBCatSearch` are

Trap macro	Selector
`_HFSDispatch`	$0018

RESULT CODES

`noErr`	0	No error (entire catalog has not been searched)
`nsvErr`	–35	Volume not found
`ioErr`	–36	I/O error
`eofErr`	–39	Logical end-of-file reached
`paramErr`	–50	Parameters don't specify an existing volume
`extFSErr`	–58	External file system
`wrgVolTypErr`	–123	Volume is an MFS volume
`catChangedErr`	–1304	Catalog has changed and catalog position record may be invalid
`afpCatalogChanged`	–5037	Catalog has changed and search cannot be resumed

SEE ALSO

See "Searching a Volume" on page 2-39 for a description of how to use `PBCatSearch`.

Exchanging the Data in Two Files

The function `PBExchangeFiles` allows you to exchange the data in two files.

PBExchangeFiles

You can use the `PBExchangeFiles` function to exchange the data stored in two files on the same volume.

```
FUNCTION PBExchangeFiles (paramBlock: HParmBlkPtr;
                          async: Boolean): OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioDestNamePtr`	`StringPtr`	A pointer to the name of the destination file.
→	`ioDestDirID`	`LongInt`	The destination file's parent directory ID.
→	`ioSrcDirID`	`LongInt`	The source file's parent directory ID.

DESCRIPTION

The `PBExchangeFiles` function swaps the data in two files by changing some of the information in the volume catalog and, if the files are open, in the file control blocks. The `PBExchangeFiles` function uses the file ID parameter block.

You should use `PBExchangeFiles` to preserve the file ID when updating an existing file, in case the file is being tracked through its file ID.

Typically, you use `PBExchangeFiles` after creating a new file during a safe save. You identify the names and parent directory IDs of the two files to be exchanged in the fields `ioNamePtr`, `ioDestNamePtr`, `ioSrcDirID`, and `ioDestDirID`. The `PBExchangeFiles` function changes the fields in the catalog entries that record the location of the data and the modification dates. It swaps both the data forks and the resource forks.

The `PBExchangeFiles` function works on either open or closed files. If either file is open, `PBExchangeFiles` updates any file control blocks associated with the file. Exchanging the contents of two files requires essentially the same access privileges as opening both files for writing.

The `PBExchangeFiles` function does not require that file ID references exist for the files being exchanged.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBExchangeFiles` are

Trap macro	Selector
`_HFSDispatch`	$0017

RESULT CODES

noErr	0	No error
nsvErr	–35	Volume not found
ioErr	–36	I/O error
fnfErr	–43	File not found
fLckdErr	–45	File is locked
vLckdErr	–46	Volume is locked or read-only
paramErr	–50	Function not supported by volume
volOfflinErr	–53	Volume is offline
wrgVolTypErr	–123	Not an HFS volume
diffVolErr	–1303	Files on different volumes
afpAccessDenied	–5000	User does not have the correct access
afpObjectTypeErr	–5025	Object is a directory, not a file
afpSameObjectErr	–5038	Source and destination are the same

Shared Environment Routines

The File Manager provides a number of routines that allow you to control access to files, directories, and volumes in a shared environment. The routines described in this section allow you to

- provide multiple users with read/write access to files
- lock and unlock portions of files opened with shared read/write permission
- manipulate share points on local shared volumes
- get and change the access privileges for directories
- mount remote volumes
- control login access
- access a list of users and groups on the local file server

Before using the routines described in this section, call the `PBHGetVolParms` function to see if the volume supports them. (The `PBGetVolMountInfoSize`, `PBGetVolMountInfo`, and `PBVolumeMount` routines are exceptions: you'll just have to make these calls and check the result code.)

Opening Files While Denying Access

The `PBHOpenDeny` and `PBHOpenRFDeny` functions control file access modes and enable applications to implement shared read/write access to files.

PBHOpenDeny

You can use the `PBHOpenDeny` function to open a file's data fork using the access deny modes.

```
FUNCTION PBHOpenDeny (paramBlock: HParmBlkPtr; async: Boolean):
                          OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
←	`ioRefNum`	`Integer`	The file reference number.
→	`ioDenyModes`	`Integer`	Access rights data.
→	`ioDirID`	`LongInt`	The directory ID.

DESCRIPTION

The `PBHOpenDeny` function opens a file's data fork with specific access rights specified in the `ioDenyModes` field. The file reference number is returned in `ioRefNum`.

The result code `opWrErr` is returned if you've requested write permission and you have already opened the file for writing; in that case, the existing file reference number is returned in `ioRefNum`. You should not use this reference number unless your application originally opened the file.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBHOpenDeny` are

Trap macro	Selector
`_HFSDispatch`	$0038

RESULT CODES

`noErr`	0	No error
`tmfoErr`	–42	Too many files open
`fnfErr`	–43	File not found
`fLckdErr`	–45	File is locked
`vLckdErr`	–46	Volume is locked or read-only
`opWrErr`	–49	File already open for writing
`paramErr`	–50	Function not supported by volume
`permErr`	–54	File is already open and cannot be opened using specified deny modes
`afpAccessDenied`	–5000	User does not have the correct access to the file
`afpDenyConflict`	–5006	Requested access permission not possible

PBHOpenRFDeny

You can use the PBHOpenRFDeny function to open a file's resource fork using the access deny modes.

```
FUNCTION PBHOpenRFDeny (paramBlock: HParmBlkPtr;
                        async: Boolean): OSErr;
```

paramBlock A pointer to a basic HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
←	ioRefNum	Integer	The file reference number.
→	ioDenyModes	Integer	Access rights data.
→	ioDirID	LongInt	The directory ID.

DESCRIPTION

The PBHOpenRFDeny function opens a file's resource fork with specific access rights. The path reference number is returned in ioRefNum.

The result code opWrErr is returned if you've requested write permission and you have already opened the file for writing; in that case, the existing file reference number is returned in ioRefNum. You should not use this reference number unless your application originally opened the file.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBHOpenRFDeny are

Trap macro	Selector
_HFSDispatch	$0039

RESULT CODES

noErr	0	No error
tmfoErr	–42	Too many files open
fnfErr	–43	File not found
fLckdErr	–45	File is locked
vLckdErr	–46	Volume is locked or read-only
opWrErr	–49	File already open for writing
paramErr	–50	Function not supported by volume
permErr	–54	File is already open and cannot be opened using specified deny modes
afpAccessDenied	–5000	User does not have the correct access to the file
afpDenyConflict	–5006	Requested access permission not possible

Locking and Unlocking File Ranges

The File Manager provides several low-level routines that allow you to lock and unlock parts of files. These functions are ineffective when used on local HFS volumes unless local file sharing is enabled for those volumes.

PBLockRange

You can use the `PBLockRange` function to lock a portion of a file.

```
FUNCTION PBLockRange (paramBlock: ParmBlkPtr; async: Boolean):
                        OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioRefNum`	`Integer`	A file reference number.
→	`ioReqCount`	`LongInt`	The number of bytes in the range.
→	`ioPosMode`	`Integer`	The positioning mode.
→	`ioPosOffset`	`LongInt`	The positioning offset.

DESCRIPTION

The `PBLockRange` function locks a portion of a file that was opened with shared read / write permission. The beginning of the range to be locked is determined by the `ioPosMode` and `ioPosOffset` fields. The end of the range to be locked is determined by the beginning of the range and the `ioReqCount` field. For example, to lock the first 50 bytes in a file, set `ioReqCount` to 50, `ioPosMode` to `fsFromStart`, and `ioPosOffset` to 0. Set `ioReqCount` to –1 to lock the maximum number of bytes from the position specified in `ioPosOffset`.

The `PBLockRange` function uses the same parameters as both `PBRead` and `PBWrite`; by calling it immediately before `PBRead`, you can use the information in the parameter block for the `PBRead` call.

When you're finished with the data (typically after a call to `PBWrite`), be sure to call `PBUnlockRange` to free that portion of the file for subsequent `PBRead` calls.

SPECIAL CONSIDERATIONS

The `PBLockRange` function does nothing if the file specified in the `ioRefNum` field is open with shared read / write permission but is not located on a remote server volume or is not located under a share point on a sharable local volume. See "Locking and

Unlocking File Ranges" on page 2-51 for a simple way to determine whether calling `PBLockRange` on an open file would in fact lock a range of bytes.

▲ **WARNING**

In system software versions 6.0.7 and earlier, specifying `ioPosMode` as `fsFromLEOF` results in the wrong byte range being locked. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBLockRange` are

Trap macro	Selector
`_HFSDispatch`	$0010

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`eofErr`	–39	Logical end-of-file reached
`fLckdErr`	–45	File is locked by another user
`paramErr`	–50	Negative `ioReqCount`
`rfNumErr`	–51	Bad reference number
`extFSErr`	–58	External file system
`volGoneErr`	–124	Server volume has been disconnected
`afpNoMoreLocks`	–5015	No more ranges can be locked
`afpRangeOverlap`	–5021	Part of range is already locked

PBUnlockRange

You can use the `PBUnlockRange` function to unlock a portion of a file that was previously locked by a call to `PBLockRange`.

```
FUNCTION PBUnlockRange (paramBlock: ParmBlkPtr; async: Boolean):
                        OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioRefNum`	`Integer`	A file reference number.
→	`ioReqCount`	`LongInt`	The number of bytes in the range.
→	`ioPosMode`	`Integer`	The positioning mode.
→	`ioPosOffset`	`LongInt`	The positioning offset.

DESCRIPTION

The `PBUnlockRange` function unlocks a portion of a file that you locked with `PBLockRange`. You specify the range by filling in the `ioReqCount`, `ioPosMode`, and `ioPosOffset` fields as described in the preceding discussion of `PBLockRange`. The range of bytes to be unlocked must be the exact same range locked by a previous call to `PBLockRange`.

If for some reason you need to unlock a range whose beginning or length is unknown, you can simply close the file. When a file is closed, all locked ranges held by the user are unlocked.

SPECIAL CONSIDERATIONS

The `PBUnlockRange` function does nothing if the file specified in the `ioRefNum` field is open with shared read/write permission but is not located on a remote server volume or is not located under a share point on a local volume.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBUnlockRange` are

Trap macro	Selector
`_HFSDispatch`	$0011

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`fnOpnErr`	–38	File not open
`eofErr`	–39	Logical end-of-file reached
`paramErr`	–50	Negative `ioReqCount`
`rfNumErr`	–51	Bad reference number
`extFSErr`	–58	External file system
`volGoneErr`	–124	Server volume has been disconnected
`afpRangeNotLocked`	–5020	Specified range was not locked

Manipulating Share Points

The `PBShare` and `PBUnshare` functions allow you to manipulate share points on local volumes. The `PBGetUGEntry` function lets you access the list of user and group names and IDs on the local server.

PBShare

You can use the PBShare function to establish a local volume or directory as a share point.

```
FUNCTION PBShare (paramBlock: HParmBlkPtr; async: Boolean): OSErr;
```

paramBlock A pointer to a basic HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioDirID	LongInt	A directory ID.

DESCRIPTION

The PBShare function makes the directory specified by the ioNamePtr and ioDirID fields a share point. If ioNamePtr is NIL, then ioDirID is the directory ID of the directory that is to become a share point. If ioNamePtr points to a partial pathname, ioDirID is the parent directory of the directory to be shared. The ioVRefNum field can contain a volume reference number, a working directory reference number, a drive number, or 0 for the default volume.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBShare are

Trap macro	Selector
_HFSDispatch	$0042

RESULT CODES

noErr	0	No error
tmfoErr	–42	Too many share points
fnfErr	–43	File not found
dupFNErr	–48	Already a share point with this name
paramErr	–50	Function not supported by volume
dirNFErr	–120	Directory not found
afpAccessDenied	–5000	This directory cannot be shared
afpObjectTypeErr	–5025	Object was a file, not a directory
afpContainsSharedErr	–5033	The directory contains a share point
afpInsideSharedErr	–5043	The directory is inside a shared directory

PBUnshare

You can use the `PBUnshare` function to reverse the effects of `PBShare`.

```
FUNCTION PBUnShare (paramBlock: HParmBlkPtr; async: Boolean):
                    OSErr;
```

`paramBlock` A pointer to a basic HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioDirID`	`LongInt`	A directory ID.

DESCRIPTION

The `PBUnshare` function makes the share point specified by the `ioNamePtr` and `ioDirID` fields unavailable on the network. If `ioNamePtr` is `NIL`, then `ioDirID` is the directory ID of the directory that is to become unavailable. If `ioNamePtr` points to a partial pathname, `ioDirID` is the parent directory of the directory to become unavailable. The `ioVRefNum` field can contain a volume reference number, a working directory reference number, a drive number, or 0 for the default volume.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBUnshare` are

Trap macro	Selector
`_HFSDispatch`	$0043

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	File not found
`paramErr`	–50	Function not supported by volume
`dirNFErr`	–120	Directory not found
`afpObjectTypeErr`	–5025	Object was a file, not a directory; or, this directory is not a share point

PBGetUGEntry

You can use the `PBGetUGEntry` function to get a list of user and group entries from the local file server.

```
FUNCTION PBGetUGEntry (paramBlock: HParmBlkPtr; async: Boolean):
                          OSErr;
```

`paramBlock` A pointer to an `objParam` variant of an HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioObjType`	`Integer`	A function code.
→	`ioObjNamePtr`	`Ptr`	A pointer to the returned user/group name.
↔	`ioObjID`	`LongInt`	A user/group ID.

DESCRIPTION

The `PBGetUGEntry` function returns the name and ID of the user or group whose name is alphabetically next to that of the user or group whose ID is contained in the `ioObjID` field. You can enumerate the users or groups in alphabetical order by setting `ioObjID` to 0 and then repetitively calling `PBGetUGEntry` with the same parameter block until the result code `fnfErr` is returned.

You specify whether you want information about users or groups by setting the `ioObjType` field to the desired value. Set `ioObjType` to 0 to receive the next user entry; set it to –1 to receive the next group entry.

The user or group name is returned as a Pascal string pointed to by `ioObjNamePtr`. The maximum size of the string is 31 characters, preceded by a length byte. If you set `ioObjNamePtr` to `NIL`, no name is returned.

If you set `ioObjID` to 0, `PBGetUGEntry` returns information about the user or group known to the local server whose name is alphabetically first. If the value of `ioObjID` is not 0, `PBGetUGEntry` returns information about the user or group whose name follows immediately in alphabetical order that of the user or group having that ID.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBGetUGEntry` are

Trap macro	Selector
`_HFSDispatch`	$0044

RESULT CODES

`noErr`	0	No error
`fnfErr`	–43	No more users or groups
`paramErr`	–50	Function not supported; or, `ioObjID` is negative

Controlling Directory Access

The PBHGetDirAccess and PBHSetDirAccess functions control privileges for individual directories.

PBHGetDirAccess

You can use the PBHGetDirAccess function to get the access control information for a directory.

```
FUNCTION PBHGetDirAccess (paramBlock: HParmBlkPtr;
                          async: Boolean): OSErr;
```

paramBlock A pointer to an HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
←	ioACOwnerID	LongInt	The owner ID.
←	ioACGroupID	LongInt	The group ID.
←	ioACAccess	LongInt	The access rights.
→	ioDirID	LongInt	The directory ID.

DESCRIPTION

The PBHGetDirAccess returns access control information for the specified directory. On output, the ioACOwnerID field contains the ID of the directory's owner, and the ioACGroupID field contains the directory's primary group. The directory's access rights are encoded in the ioACAccess field. See "Directory Access Privileges," beginning on page 2-18, for a description of the ioACAccess field.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBHGetDirAccess are

Trap macro	Selector
_HFSDispatch	$0032

RESULT CODES

noErr	0	No error
fnfErr	–43	Directory not found
paramErr	–50	Function not supported by volume
afpAccessDenied	–5000	User does not have the correct access to the directory

PBHSetDirAccess

You can use the PBHSetDirAccess function to change the access control information for a directory.

```
FUNCTION PBHSetDirAccess (paramBlock: HParmBlkPtr;
                          async: Boolean): OSErr;
```

paramBlock A pointer to an HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioACOwnerID	LongInt	The owner ID.
→	ioACGroupID	LongInt	The group ID.
→	ioACAccess	LongInt	The access rights.
→	ioDirID	LongInt	The directory ID.

DESCRIPTION

The PBHSetDirAccess function allows you to change the access rights to the specified directory. The ioACAccess field contains the directory's access rights. You cannot set the owner or user rights bits of the ioACAccess field directly (if you try to do this, PBHSetDirAccess returns the result code paramErr). See "Directory Access Privileges," beginning on page 2-18, for a description of the ioACAccess field.

To change the owner or group, you should set the ioACOwnerID or ioACGroupID field to the appropriate ID. You must be the owner of the directory to change the owner or group ID. A guest on a server can manipulate the privileges of any directory owned by the guest.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBHSetDirAccess are

Trap macro	Selector
_HFSDispatch	$0033

RESULT CODES

noErr	0	No error
fnfErr	–43	Directory not found
vLckdErr	–46	Volume is locked or read-only
paramErr	–50	Parameter error
afpAccessDenied	–5000	User does not have the correct access to the directory
afpObjectTypeErr	–5025	Object is a file, not a directory

Mounting Volumes

The File Manager provides three functions that allow your application to record the mounting information for a volume and then to mount the volume later. The programmatic mounting functions store the mounting information in a structure called the `AFPVolMountInfo` record. The programmatic mounting functions use the `ioParam` variant of the `ParamBlockRec` record.

In general, it is easier to mount remote volumes by creating and then resolving alias records that describe those volumes. The Alias Manager displays the standard user interface for user authentication when resolving alias records for remote volumes. As a result, the routines described in this section are primarily of interest for applications that need to mount remote volumes with no user interface or with some custom user interface.

Note
All the functions described in this section execute synchronously. You should not call them at interrupt time. ◆

PBGetVolMountInfoSize

You use the `PBGetVolMountInfoSize` function to determine how much space to allocate for a volume mounting information record.

```
FUNCTION PBGetVolMountInfoSize (paramBlock: ParmBlkPtr): OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

Parameter block

→	`ioCompletion`	`LongInt`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The function's result code.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioBuffer`	`LongInt`	A pointer to storage for size.

DESCRIPTION

For a specified volume, the `PBGetVolMountInfoSize` function provides the size of the record needed to hold the volume's mounting information. The `ioBuffer` field is a pointer to the size information, which is of type `Integer` (2 bytes). If `PBGetVolMountInfoSize` returns `noErr`, that integer contains the size of the volume mounting information record.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBGetVolMountInfoSize` are

Trap macro	Selector
`_HFSDispatch`	$003F

RESULT CODES

noErr	0	No error
nsvErr	–35	Volume not found
paramErr	–50	Parameter error
extFSErr	–58	External file system error; typically, function is not available for that volume

PBGetVolMountInfo

After ascertaining the size of the record needed and allocating storage, you can use the PBGetVolMountInfo function to retrieve a record containing all the information needed to mount the volume, except for passwords. You can later pass this record to the PBVolumeMount function to mount the volume.

```
FUNCTION PBGetVolMountInfo (paramBlock: ParmBlkPtr): OSErr;
```

paramBlock A pointer to a basic File Manager parameter block.

Parameter block

→	ioCompletion	LongInt	A pointer to a completion routine.
←	ioResult	OSErr	The function's result code.
→	ioVRefNum	Integer	A volume specification.
→	ioBuffer	LongInt	A pointer to mounting information.

DESCRIPTION

The PBGetVolMountInfo function places the mounting information for a specified volume into the buffer pointed to by the ioBuffer field. The mounting information for an AppleShare volume is stored as an AFP mounting record. The length of the buffer is specified by the value pointed to by the ioBuffer field in a previous call to PBGetVolMountInfoSize.

The PBGetVolMountInfo function does not return the user password or volume password in the AFPVolMountInfo record. Your application should solicit these passwords from the user and fill in the record before attempting to mount the remote volume.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBGetVolMountInfo are

Trap macro	Selector
_HFSDispatch	$0040

RESULT CODES

noErr	0	No error
nsvErr	–35	Volume not found
paramErr	–50	Parameter error
extFSErr	–58	External file system error; typically, function is not available for that volume

PBVolumeMount

You can use the `PBVolumeMount` function to mount a volume, using either the information returned by the `PBGetVolMountInfo` function or a structure filled in by your application.

```
FUNCTION PBVolumeMount (paramBlock: ParmBlkPtr): OSErr;
```

`paramBlock` A pointer to a basic File Manager parameter block.

Parameter block

→	ioCompletion	LongInt	A pointer to a completion routine.
←	ioResult	OSErr	The function's result code.
←	ioVRefNum	Integer	A volume reference number.
→	ioBuffer	LongInt	A pointer to mounting information.

DESCRIPTION

The `PBVolumeMount` function mounts a volume and returns its volume reference number. If you're mounting an AppleShare volume, place the volume's AFP mounting information record in the buffer pointed to by the `ioBuffer` field.

The `PBGetVolMountInfo` function does not return the user and volume passwords; they're returned blank. Typically, your application asks the user for any necessary passwords and fills in those fields just before calling `PBVolumeMount`. If you want to mount a volume with guest status, pass an empty string as the user password.

If you have enough information about the volume, you can fill in the mounting record yourself and call `PBVolumeMount`, even if you did not save the mounting information while the volume was mounted. To mount an AFP volume, you must fill in the record with at least the zone name, server name, user name, user password, and volume password. You can lay out the fields in any order within the data field, as long as you specify the correct offsets.

SPECIAL CONSIDERATIONS

The File Sharing workstation software introduced in system software version 7.0 does not currently pass the volume password. The AppleShare 3.0 workstation software does, however, pass the volume password.

AFP volumes currently ignore the user authentication method passed in the `uamType` field of the volume mounting information record whose address is passed in `ioBuffer`. The most secure available method is used by default, except when a user mounts the volume as <Guest> and uses the `kNoUserAuthentication` authentication method.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBVolumeMount` are

Trap macro	Selector
`_HFSDispatch`	$0041

RESULT CODES

`noErr`	0	No error
`notOpenErr`	–28	AppleTalk is not open
`nsvErr`	–35	Volume not found
`paramErr`	–50	Parameter error; typically, zone, server, and volume name combination is not valid or not complete, or the user name is not recognized
`extFSErr`	–58	External file system error; typically, file system signature was not recognized, or function is not available for that volume
`memFullErr`	–108	Not enough memory to create a new volume control block for mounting the volume
`afpBadUAM`	–5002	User authentication method is unknown
`afpBadVersNum`	–5003	Workstation is using an AFP version that the server doesn't recognize
`afpNoServer`	–5016	Server is not responding
`afpUserNotAuth`	–5023	User authentication failed (usually, password is not correct)
`afpPwdExpired`	–5042	Password has expired on server
`afpBadDirIDType`	–5060	Not a fixed directory ID volume
`afpCantMountMoreSrvrs`	–5061	Maximum number of volumes has been mounted
`afpAlreadyMounted`	–5062	Volume already mounted
`afpSameNodeErr`	–5063	Attempt to log on to a server running on the same machine

Controlling Login Access

You can use the functions `PBHGetLogInInfo`, `PBHMapID`, and `PBHMapName` to get information about the login method and the recognized users and groups on a particular machine.

PBHGetLogInInfo

You can use the `PBHGetLogInInfo` function to determine the login method used to log on to a particular shared volume.

```
FUNCTION PBHGetLogInInfo (paramBlock: HParmBlkPtr;
                          async: Boolean): OSErr;
```

`paramBlock` A pointer to an `objParam` variant of the HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioVRefNum`	`Integer`	The volume specification.
←	`ioObjType`	`Integer`	The login method.
←	`ioObjNamePtr`	`Ptr`	A pointer to the user name.

DESCRIPTION

The `PBHGetLogInInfo` function returns the method used for login and the user name specified at login time for the volume specified by the `ioVRefNum` field. The login user name is returned as a Pascal string in `ioObjNamePtr`. The maximum size of the user name is 31 characters. The login method type is returned in the `ioObjType` field. These values are recognized.

```
CONST
   kNoUserAuthentication   = 1;  {no password}
   kPassword               = 2;  {8-byte password}
   kEncryptPassword        = 3;  {encrypted 8-byte password}
   kTwoWayEncryptPassword  = 6;  {two-way random encryption}
```

Values in the range 7–127 are reserved for future use by Apple Computer, Inc. Values in the range 128–255 are available to your application as user-defined values.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBHGetLogInInfo` are

Trap macro	Selector
`_HFSDispatch`	$0031

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Specified volume doesn't exist
`paramErr`	–50	Function not supported by volume

PBHMapID

You can use the PBHMapID function to determine the name of a user or group if you know the user or group ID.

```
FUNCTION PBHMapID (paramBlock: HParmBlkPtr; async: Boolean):
                    OSErr;
```

paramBlock A pointer to an objParam variant of the HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioObjType	Integer	The login method.
←	ioObjNamePtr	Ptr	A pointer to the user / group name.
→	ioObjID	LongInt	The user / group ID.

DESCRIPTION

The PBHMapID function returns the name of a user or group given its unique ID. The ioObjID field contains the ID to be mapped. (AppleShare uses the value 0 to signify <Any User>.) The ioObjType field is the mapping function code; its value is 1 if you're mapping a user ID to a user name or 2 if you're mapping a group ID to a group name. The name is returned in ioObjNamePtr; the maximum size of the name is 31 characters (preceded by a length byte).

Because user and group IDs are interchangeable under AFP 2.1 and later volumes, you might not need to specify a value in the ioObjType field.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBHMapID are

Trap macro	Selector
_HFSDispatch	$0034

RESULT CODES

noErr	0	No error
fnfErr	–43	Unrecognizable owner or group name
paramErr	–50	Function not supported by volume

PBHMapName

You can use the PBHMapName function to determine the user ID or group ID from a user or group name.

```
FUNCTION PBHMapName (paramBlock: HParmBlkPtr; async: Boolean):
                     OSErr;
```

paramBlock A pointer to an objParam variant of the HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioObjType	Integer	The login method.
→	ioObjNamePtr	Ptr	A pointer to the user/group name.
←	ioObjID	LongInt	The user/group ID.

DESCRIPTION

Given a name, the PBHMapName function returns the corresponding unique user ID or group ID. The name is passed as a string in ioObjNamePtr. If NIL is passed, the ID returned is always 0. The maximum size of the name is 31 characters. The ioObjType field is the mapping function code; its value is 3 if you're mapping a user name to a user ID or 4 if you're mapping a group name to a group ID. On exit, ioObjID contains the mapped ID.

Because user and group IDs are interchangeable under AFP 2.1 and later volumes, you might need to set the ioObjType field to determine which database (user or group) to search first. If both a user and a group have the same name, this field determines which kind of ID you receive.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBHMapName are

Trap macro	Selector
_HFSDispatch	$0035

RESULT CODES

noErr	0	No error
fnfErr	–43	Unrecognizable owner or group name
paramErr	–50	Function not supported by volume

Copying and Moving Files

The File Manager provides two shared environment routines—PBHCopyFile and PBHMoveRename—that allow you to copy and move files. These routines are especially useful when you want to copy or move files located on a remote volume, because they allow you to forgo transmitting large amounts of data across a network. These routines are used internally by the Finder; most applications do not need to use them.

If you do want to use PBHCopyFile or PBHMoveRename, you should first call PBHGetVolParms to see whether the target volume supports these routines.

PBHCopyFile

You can use the PBHCopyFile function to duplicate a file and optionally to rename it.

```
FUNCTION PBHCopyFile (paramBlock: HParmBlkPtr; async: Boolean):
                        OSErr;
```

paramBlock A pointer to a copyParam variant of the HFS parameter block.

async A Boolean value that specifies asynchronous (TRUE) or synchronous (FALSE) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioDstVRefNum	Integer	Destination volume identifier.
→	ioNewName	Ptr	A pointer to the destination pathname (may be NIL).
→	ioCopyName	Ptr	A pointer to the file's new name (may be NIL).
→	ioNewDirID	LongInt	The destination directory ID.
→	ioDirID	LongInt	The source directory ID.

DESCRIPTION

The PBHCopyFile function duplicates a file on the specified volume and optionally renames it. It is an optional call for AppleShare file servers. Your application should examine the information returned by the PBHGetVolParms function to see if the volume supports PBHCopyFile.

For AppleShare file servers, the source and destination pathnames must indicate the same file server; however, the parameter block may specify different source and destination volumes on that file server. A useful way to tell if two file server volumes are on the same file server is to call the PBHGetVolParms function for each volume and compare the server addresses returned. The server opens source files with read / deny write enabled and destination files with write / deny read and write enabled.

You specify the source file with the `ioVRefNum`, `ioDirID`, and `ioNamePtr` fields. You specify the destination directory with the `ioDstVRefNum`, `ioNewDirID`, and `ioNewName` fields. If `ioNewName` is `NIL`, the destination directory is the directory having ID `ioNewDirID` on the specified volume; if `ioNewName` is not `NIL`, the destination directory is the directory having the partial pathname pointed to by `ioNewName` in the directory having ID `ioNewDirID` on the specified volume.

The `ioCopyName` field may contain a pointer to an optional string to be used in copying the file; if it is not `NIL`, the file copy is renamed to the name specified in `ioCopyName`. The string pointed to by `ioCopyName` must be a filename, not a partial pathname.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBHCopyFile` are

Trap macro	Selector
`_HFSDispatch`	$0036

RESULT CODES

`noErr`	0	No error
`dskFulErr`	–34	Destination volume is full
`fnfErr`	–43	Source file not found, or destination directory does not exist
`vLckdErr`	–46	Destination volume is read-only
`fBsyErr`	–47	The source or destination file could not be opened with the correct access modes
`dupFNErr`	–48	Destination file already exists
`paramErr`	–50	Function not supported by volume
`wrgVolTypErr`	–123	Function not supported by volume
`afpAccessDenied`	–5000	The user does not have the right to read the source or write to the destination
`afpDenyConflict`	–5006	The source or destination file could not be opened with the correct access modes
`afpObjectTypeErr`	–5025	Source is a directory

PBHMoveRename

You can use the `PBHMoveRename` function to move a file or directory and optionally to rename it.

```
FUNCTION PBHMoveRename (paramBlock: HParmBlkPtr;
                        async: Boolean): OSErr;
```

`paramBlock` A pointer to a `copyParam` variant of the HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	ioCompletion	ProcPtr	A pointer to a completion routine.
←	ioResult	OSErr	The result code of the function.
→	ioNamePtr	StringPtr	A pointer to a pathname.
→	ioVRefNum	Integer	A volume specification.
→	ioNewName	Ptr	A pointer to the destination pathname (may be NIL).
→	ioCopyName	Ptr	A pointer to the file's new name (may be NIL).
→	ioNewDirID	LongInt	The destination directory ID.
→	ioDirID	LongInt	The source directory ID.

DESCRIPTION

The PBHMoveRename function allows you to move (not copy) a file or directory and optionally to rename it. The source and destination pathnames must point to the same file server volume.

You specify the source file or directory with the ioVRefNum, ioDirID, and ioNamePtr fields. You specify the destination directory with the ioNewDirID and ioNewName fields. If ioNewName is NIL, the destination directory is the directory having ID ioNewDirID on the specified volume; if ioNewName is not NIL, the destination directory is the directory having the partial pathname pointed to by ioNewName in the directory having ID ioNewDirID on the specified volume.

The ioCopyName field may contain a pointer to an optional string to be used in copying the file or directory; if it is not NIL, the moved object is renamed to the name specified in ioCopyName. The string pointed to by ioCopyName must be a filename, not a partial pathname.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for PBHMoveRename are

Trap macro	Selector
_HFSDispatch	$0037

RESULT CODES

noErr	0	No error
fnfErr	–43	Source file or directory not found
fLckdErr	–45	File is locked
vLckdErr	–46	Destination volume is read-only
dupFNErr	–48	Destination already exists
paramErr	–50	Function not supported by volume
badMovErr	–122	Attempted to move directory into offspring
afpAccessDenied	–5000	The user does not have the right to move the file or directory

File ID Routines

The File Manager provides several routines that allow you to track files using file IDs. These routines use the `fidParam` variant of the HFS parameter block.

Note

Most applications do not need to use these routines. In general you should track files using alias records, as described in the chapter "Alias Manager" in this book. The Alias Manager uses file IDs internally as part of its search algorithms for finding the target of an alias record. ◆

Resolving File ID References

You can find the target of a file ID reference by calling the `PBResolveFileIDRef` function.

PBResolveFileIDRef

You can use the `PBResolveFileIDRef` function to retrieve the filename and parent directory ID of the file with a specified file ID.

```
FUNCTION PBResolveFileIDRef (paramBlock: HParmBlkPtr;
                             async: Boolean): OSErr;
```

`paramBlock` A pointer to an `fidParam` variant of the HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
↔	`ioNamePtr`	`StringPtr`	A pointer to a filename.
→	`ioVRefNum`	`Integer`	A volume specification.
←	`ioSrcDirID`	`LongInt`	The file's parent directory ID.
→	`ioFileID`	`LongInt`	A file ID.

DESCRIPTION

The `PBResolveFileIDRef` function returns the filename and parent directory ID of the file referred to by file ID in the `ioFileID` field. It places the filename in the string pointed to by the `ioNamePtr` field and the parent directory ID in the `ioSrcDirID` field. If the name string is `NIL`, `PBResolveFileIDRef` returns only the parent directory ID. If the name string is not `NIL` but is only a volume name, `PBResolveFileIDRef` ignores the value in the `ioVRefNum` field, uses the volume name instead, and overwrites the name string with the filename. A return code of `fidNotFoundErr` means that the specified file ID reference has become invalid, either because the file was deleted or because the file ID reference was destroyed by `PBDeleteFileIDRef`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBResolveFileIDRef` are

Trap macro	Selector
`_HFSDispatch`	$0016

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`ioErr`	–36	I/O error
`fnfErr`	–43	File not found
`paramErr`	–50	Function not supported by volume
`volOfflinErr`	–53	Volume is offline
`extFSErr`	–58	External file system
`wrgVolTypErr`	–123	Not an HFS volume
`fidNotFoundErr`	–1300	File ID not found
`notAFileErr`	–1302	Specified file is a directory
`afpAccessDenied`	–5000	User does not have the correct access
`afpObjectTypeErr`	–5025	Specified file is a directory
`afpIDNotFound`	–5034	File ID not found
`afpBadIDErr`	–5039	File ID not found

Creating and Deleting File ID References

You can create and delete file ID references using the functions `PBCreateFileIDRef` and `PBDeleteFileIDRef`.

Note

Most applications should not directly create or delete file ID references. ◆

PBCreateFileIDRef

Use the `PBCreateFileIDRef` function to establish a file ID reference for a file.

```
FUNCTION PBCreateFileIDRef (paramBlock: HParmBlkPtr;
                            async: Boolean): OSErr;
```

`paramBlock` A pointer to an `fidParam` variant of the HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a filename.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioSrcDirID`	`LongInt`	The file's parent directory ID.
←	`ioFileID`	`LongInt`	A file ID.

DESCRIPTION

Given a volume reference number, filename, and parent directory ID, the `PBCreateFileIDRef` function creates a record to hold the name and parent directory ID of the specified file. `PBCreateFileIDRef` places the file ID in the `ioFileID` field. If a file ID reference already exists for the file, `PBCreateFileIDRef` supplies the file ID but returns the result code `fidExists`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBCreateFileIDRef` are

Trap macro	Selector
`_HFSDispatch`	$0014

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`ioErr`	–36	I/O error
`fnfErr`	–43	File not found
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`paramErr`	–50	Function not supported by volume
`volOfflinErr`	–53	Volume is offline
`extFSErr`	–58	External file system
`wrgVolTypErr`	–123	Not an HFS volume
`fidExists`	–1301	File ID already exists
`notAFileErr`	–1302	Specified file is a directory
`afpAccessDenied`	–5000	User does not have the correct access
`afpObjectTypeErr`	–5025	Specified file is a directory
`afpIDExists`	–5035	File ID already exists

PBDeleteFileIDRef

You can use the `PBDeleteFileIDRef` function to delete a file ID reference.

```
FUNCTION PBDeleteFileIDRef (paramBlock: HParmBlkPtr;
                            async: Boolean): OSErr;
```

`paramBlock` A pointer to an `fidParam` variant of the HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a filename.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioFileID`	`LongInt`	A file ID.

DESCRIPTION

The `PBDeleteFileIDRef` function invalidates the specified file ID reference on the volume specified by `ioVRefNum` or `ioNamePtr`. After it has invalidated a file ID reference, the File Manager can no longer resolve that ID reference to a filename and parent directory ID.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBDeleteFileIDRef` are

Trap macro	Selector
`_HFSDispatch`	$0015

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`ioErr`	–36	I/O error
`fnfErr`	–43	File not found
`wPrErr`	–44	Hardware volume lock
`vLckdErr`	–46	Software volume lock
`paramErr`	–50	Function not supported by volume
`volOfflinErr`	–53	Volume is offline
`extFSErr`	–58	External file system
`wrgVolTypErr`	–123	Function is not supported by volume
`fidNotFoundErr`	–1300	File ID not found
`afpAccessDenied`	–5000	User does not have the correct access
`afpObjectTypeErr`	–5025	Specified file is a directory
`afpIDNotFound`	–5034	File ID not found

Foreign File System Routines

The File Manager provides several routines that allow you to obtain and set privilege information on foreign file systems. The `PBGetForeignPrivs` and `PBSetForeignPrivs` functions allow your application or shell program to communicate with a foreign file system about its native access-control system. These functions retrieve and set access permissions on the foreign file system, using a `foreignPrivParam` variant of the HFS parameter block.

PBGetForeignPrivs

You can use the `PBGetForeignPrivs` function to determine the native access-control information for a file or directory stored on a volume managed by a foreign file system.

```
FUNCTION PBGetForeignPrivs (paramBlock: HParmBlkPtr;
                            async: Boolean): OSErr;
```

`paramBlock` A pointer to a `foreignPrivParam` variant of the HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a file or directory name.
←	`ioVRefNum`	`Integer`	A volume specification.
←	`ioForeignPrivBuffer`	`Ptr`	A pointer to the privilege information buffer.
→	`ioForeignPrivReqCount`	`LongInt`	The size allocated for the buffer.
←	`ioForeignPrivActCount`	`LongInt`	The amount used in buffer.
→	`ioForeignPrivDirID`	`Integer`	The parent directory ID.
←	`ioForeignPrivInfo1`	`LongInt`	Information specific to privilege model.
←	`ioForeignPrivInfo2`	`LongInt`	Information specific to privilege model.
←	`ioForeignPrivInfo3`	`LongInt`	Information specific to privilege model.
←	`ioForeignPrivInfo4`	`LongInt`	Information specific to privilege model.

DESCRIPTION

The `PBGetForeignPrivs` function retrieves access information for a file or directory on a volume managed by a file system that uses a privilege model different from the AFP model. See “Privilege Information in Foreign File Systems” on page 2-20 for a more complete explanation of access-control privileges.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBGetForeignPrivs` are

Trap macro	Selector
`_HFSDispatch`	$0060

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`paramErr`	–50	Volume is HFS or MFS (that is, it has no foreign privilege model), or foreign volume does not support these calls

PBSetForeignPrivs

You can use the `PBSetForeignPrivs` function to change the native access-control information for a file or directory stored on a volume managed by a foreign file system.

```
FUNCTION PBSetForeignPrivs (paramBlock: HParmBlkPtr;
                            async: Boolean): OSErr;
```

`paramBlock` A pointer to a `foreignPrivParam` variant of the HFS parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
→	`ioNamePtr`	`StringPtr`	A pointer to a file or directory name.
→	`ioVRefNum`	`Integer`	A volume specification.
→	`ioForeignPrivBuffer`	`Ptr`	A pointer to the privilege information buffer.
→	`ioForeignPrivReqCount`	`LongInt`	The size allocated for the buffer.
→	`ioForeignPrivActCount`	`LongInt`	The amount used in buffer.
→	`ioForeignPrivDirID`	`Integer`	The parent directory ID.
→	`ioForeignPrivInfo1`	`LongInt`	Information specific to privilege model.
→	`ioForeignPrivInfo2`	`LongInt`	Information specific to privilege model.
→	`ioForeignPrivInfo3`	`LongInt`	Information specific to privilege model.
→	`ioForeignPrivInfo4`	`LongInt`	Information specific to privilege model.

DESCRIPTION

The `PBSetForeignPrivs` function modifies access information for a file or directory on a volume managed by a file system that uses a privilege model different from the AFP model.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBSetForeignPrivs` are

Trap macro	Selector
`_HFSDispatch`	$0061

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Volume not found
`paramErr`	–50	Volume is HFS or MFS (that is, it has no foreign privilege model), or foreign volume does not support these calls

Utility Routines

The File Manager provides several utility routines that allow you to obtain information about File Manager queues and file control blocks. These routines insulate your application from the need to know about the data structures maintained internally by the File Manager. Most applications do not need to use these routines.

Obtaining Queue Headers

You can use the functions `GetFSQHdr`, `GetVCBQHdr`, and `GetDrvQHdr` to obtain a pointer to the header of the file I/O queue, the VCB queue, and the drive queue, respectively. See the chapter "Queue Utilities" in *Inside Macintosh: Operating System Utilities* for a description of queues and the format of a queue header.

GetFSQHdr

You can use the `GetFSQHdr` function to get a pointer to the header of the file I/O queue.

```
FUNCTION GetFSQHdr: QHdrPtr;
```

DESCRIPTION

The `GetFSQHdr` function returns a pointer to the header of the file I/O queue.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `FSQHdr` contains the header of the file I/O queue.

GetVCBQHdr

You can use the `GetVCBQHdr` function to get a pointer to the header of the VCB queue.

```
FUNCTION GetVCBQHdr: QHdrPtr;
```

DESCRIPTION

The `GetVCBQHdr` function returns a pointer to the header of the VCB queue.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `VCBQHdr` contains the header of the VCB queue. The default volume's VCB is pointed to by the global variable `DefVCBPtr`.

GetDrvQHdr

You can use the `GetDrvQHdr` function to get a pointer to the header of the drive queue.

```
FUNCTION GetDrvQHdr: QHdrPtr;
```

DESCRIPTION

The `GetDrvQHdr` function returns a pointer to the header of the drive queue.

ASSEMBLY-LANGUAGE INFORMATION

The global variable `DrvQHdr` contains the header of the drive queue.

Adding a Drive

The `AddDrive` procedure allows you to add a drive.

AddDrive

You can use the `AddDrive` procedure to add a drive to the system.

```
PROCEDURE AddDrive (drvrRefNum: Integer; drvNum: Integer;
                    qEl: DrvQElPtr);
```

`drvrRefNum`	A driver reference number.
`drvNum`	A drive number.
`qEl`	A pointer to a drive queue element.

DESCRIPTION

The `AddDrive` procedure adds a disk drive having the specified driver reference number and drive number to the system. The File Manager expands the drive queue by adding a copy of the queue element pointed to by the `qEl` parameter to the end of the existing queue.

Obtaining File Control Block Information

You can get information from the file control block (FCB) allocated for an open file by calling the function `PBGetFCBInfo`.

PBGetFCBInfo

You can use `PBGetFCBInfo` to get information about an open file.

```
FUNCTION PBGetFCBInfo (paramBlock: FCBPBPtr; async: Boolean):
                       OSErr;
```

`paramBlock` A pointer to a file control block parameter block.

`async` A Boolean value that specifies asynchronous (`TRUE`) or synchronous (`FALSE`) execution.

Parameter block

→	`ioCompletion`	`ProcPtr`	A pointer to a completion routine.
←	`ioResult`	`OSErr`	The result code of the function.
↔	`ioNamePtr`	`StringPtr`	A pointer to a pathname.
→	`ioVRefNum`	`Integer`	A volume specification.
↔	`ioRefNum`	`Integer`	The file reference number.
→	`ioFCBIndx`	`Integer`	An index.
←	`ioFCBFlNm`	`LongInt`	The file ID.
←	`ioFCBFlags`	`Integer`	File status flags.
←	`ioFCBStBlk`	`Integer`	The first allocation block of the file.
←	`ioFCBEOF`	`LongInt`	The logical end-of-file.
←	`ioFCBPLen`	`LongInt`	The physical end-of-file.
←	`ioFCBCrPs`	`LongInt`	The position of the file mark.
←	`ioFCBVRefNum`	`Integer`	The volume reference number.
←	`ioFCBClpSiz`	`LongInt`	The file clump size.
←	`ioFCBParID`	`LongInt`	The parent directory ID.

DESCRIPTION

The `PBGetFCBInfo` function returns information about the specified open file. If the value of `ioFCBIndx` is positive, the File Manager returns information about the file whose index in the FCB buffer is `ioFCBIndx` and that is located on the volume specified by `ioVRefNum` (which may contain a drive number, volume reference number, or working directory reference number). If the value of `ioVRefNum` is 0, all open files are indexed; otherwise, only open files on the specified volume are indexed.

If the value of `ioFCBIndx` is 0, the File Manager returns information about the file whose file reference number is specified by the `ioRefNum` field. If the value of `ioFCBIndx` is positive, the `ioRefNum` field is ignored on input and contains the file reference number on output.

If `PBGetFCBInfo` executes successfully, the `ioNamePtr` field contains the name of the specified open file. You should pass a pointer to a `Str31` value if you want that name returned. If you pass `NIL` in the `ioNamePtr` field, no filename is returned.

The `ioFCBFlags` field returns status information about the specified open file. See “File Control Block Parameter Blocks” beginning on page 2-108 for a description of the meaning of the bits in this field.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `PBGetFCBInfo` are

Trap macro	Selector
`_HFSDispatch`	$0008

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	Specified volume doesn't exist
`fnOpnErr`	–38	File not open
`rfNumErr`	–51	Reference number specifies nonexistent access path

Application-Defined Routines

This section describes the application-defined routines whose addresses you pass to some of the File Manager routines. You can define a routine that is called after the completion of an asynchronous call.

Completion Routines

Most low-level File Manager routines can be executed either synchronously (that is, the application can't continue until the routine is completed) or asynchronously (that is, the application is free to perform other tasks while the routine is executing). Some routines, however, can only be executed synchronously because they use the Memory Manager to allocate and release memory.

When you execute a routine asynchronously, you can specify a completion routine that the File Manager executes after the completion of the call.

MyCompletionProc

A File Manager completion routine has the following syntax:

```
PROCEDURE MyCompletionProc;
```

DESCRIPTION

When you execute a File Manager routine asynchronously (by setting its `async` parameter to `TRUE`), you can specify a completion routine by passing the routine's address in the `ioCompletion` field of the parameter block passed to the routine. Because you requested asynchronous execution, the File Manager places an I/O request in the file I/O queue and returns control to your application—possibly even before the actual I/O operation is completed. The File Manager takes requests from the queue one at a time and processes them; meanwhile, your application is free to do other processing.

A routine executed asynchronously returns control to your application with the result code `noErr` as soon as the call is placed in the file I/O queue. This result code does not indicate that the call has successfully completed, but simply indicates that the call was successfully placed in the queue. To determine when the call is actually completed, you can inspect the `ioResult` field of the parameter block. This field is set to a positive number when the call is made and set to the actual result code when the call is completed. If you specify a completion routine, it is executed after the result code is placed in `ioResult`.

ASSEMBLY-LANGUAGE INFORMATION

When your completion routine is called, register A0 contains a pointer to the parameter block of the asynchronous call, and register D0 contains the result code. The value in register D0 is always identical to the value in the `ioResult` field of the parameter block.

A completion routine must preserve all registers other than A0, A1, and D0–D2.

SPECIAL CONSIDERATIONS

Because a completion routine is executed at interrupt time, it should not allocate, move, or purge memory (either directly or indirectly) and should not depend on the validity of handles to unlocked blocks.

If your completion routine uses application global variables, it must also ensure that register A5 contains the address of the boundary between your application global variables and your application parameters. For details, see the discussion of the functions `SetCurrentA5` and `SetA5` in the chapter "Memory Management Utilities" in *Inside Macintosh: Memory*.

SEE ALSO

For a more complete discussion of interrupt-level processing and its limitations, see the chapter "Introduction to Processes and Tasks" in *Inside Macintosh: Processes*.

Summary of the File Manager

Pascal Summary

Constants

```
CONST
   {Gestalt constants}
   gestaltFSAttr              =  'fs  ';  {file system attributes selector}
   gestaltFullExtFSDispatching=  0;       {exports HFSDispatch traps}
   gestaltHasFSSpecCalls      =  1;       {supports FSSpec records}

   {directory IDs}
   fsRtParID         =  1;       {directory ID of root directory's parent}
   fsRtDirID         =  2;       {directory ID of volume's root directory}

   {access modes for opening files}
   fsCurPerm         =  0;       {whatever permission is allowed}
   fsRdPerm          =  1;       {read permission}
   fsWrPerm          =  2;       {write permission}
   fsRdWrPerm        =  3;       {exclusive read/write permission}
   fsRdWrShPerm      =  4;       {shared read/write permission}

   {file mark positioning modes}
   fsAtMark          =  0;       {at current mark}
   fsFromStart       =  1;       {set mark relative to beginning of file}
   fsFromLEOF        =  2;       {set mark relative to logical end-of-file}
   fsFromMark        =  3;       {set mark relative to current mark}
   rdVerify          =  64;      {add to above for read-verify}

   {values for ioSearchBits in PBCatSearch parameter block}
   fsSBPartialName   =  1;       {substring of name}
   fsSBFullName      =  2;       {full name}
   fsSBFlAttrib      =  4;       {directory flag; software lock flag}
   fsSBNegate        =  16384;   {reverse match status}
   {for files only}
   fsSBFlFndrInfo    =  8;       {Finder file info}
   fsSBFlLgLen       =  32;      {logical length of data fork}
   fsSBFlPyLen       =  64;      {physical length of data fork}
```

```
fsSBFlRLgLen      =  128;     {logical length of resource fork}
fsSBFlRPyLen      =  256;     {physical length of resource fork}
fsSBFlCrDat       =  512;     {file creation date}
fsSBFlMdDat       =  1024;    {file modification date}
fsSBFlBkDat       =  2048;    {file backup date}
fsSBFlXFndrInfo   =  4096;    {more Finder file info}
fsSBFlParID       =  8192;    {file's parent ID}
{for directories only}
fsSBDrUsrWds      =  8;       {Finder directory info}
fsSBDrNmFls       =  16;      {number of files in directory}
fsSBDrCrDat       =  512;     {directory creation date}
fsSBDrMdDat       =  1024;    {directory modification date}
fsSBDrBkDat       =  2048;    {directory backup date}
fsSBDrFndrInfo    =  4096;    {more Finder directory info}
fsSBDrParID       =  8192;    {directory's parent ID}


{value of vMForeignPrivID in file attributes buffer}
fsUnixPriv        =  1;       {A/UX privilege model}

{bit positions in vMAttrib field of GetVolParmsInfoBuffer}
bHasBlankAccessPrivileges
                  =  4;       {volume supports inherited privileges}
bHasBTreeMgr      =  5;       {reserved}
bHasFileIDs       =  6;       {volume supports file ID functions}
bHasCatSearch     =  7;       {volume supports PBCatSearch}
bHasUserGroupList =  8;       {volume supports AFP privileges}
bHasPersonalAccessPrivileges
                  =  9;       {local file sharing is enabled}
bHasFolderLock    =  10;      {volume supports locking of folders}
bHasShortName     =  11;      {volume supports AFP short names}
bHasDesktopMgr    =  12;      {volume supports Desktop Manager}
bHasMoveRename    =  13;      {volume supports _MoveRename}
bHasCopyFile      =  14;      {volume supports _CopyFile}
bHasOpenDeny      =  15;      {volume supports shared access modes}
bHasExtFSVol      =  16;      {volume is external file system volume}
bNoSysDir         =  17;      {volume has no system directory}
bAccessCntl       =  18;      {volume supports AFP access control}
bNoBootBlks       =  19;      {volume is not a startup volume}
bNoDeskItems      =  20;      {do not place objects on the desktop}
bNoSwitchTo       =  25;      {do not switch launch to applications}
bTrshOffLine      =  26;      {zoom volume when it is unmounted}
bNoLclSync        =  27;      {don't let Finder change mod. date}
bNoVNEdit         =  28;      {lock volume name}
```

```
   bNoMiniFndr       =  29;      {reserved; always 1}
   bLocalWList       =  30;      {use shared volume handle for window list}
   bLimitFCBs        =  31;      {limit file control blocks}

   {media type in remote mounting information}
   AppleShareMediaType
                     = 'afpm';   {an AppleShare volume}

   {user authentication methods in AFP remote mounting information}
   kNoUserAuthentication   =  1;    {guest status; no password needed}
   kPassword               =  2;    {8-byte password}
   kEncryptPassword        =  3;    {encrypted 8-byte password}
   kTwoWayEncryptPassword  =  6;    {two-way random encryption; }
                                    { authenticate both user and server}
```

Data Types

File System Specification Record

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

   FSSpecPtr         =  ^FSSpec;
   FSSpecHandle      =  ^FSSpecPtr;

   FSSpecArray       =  ARRAY[0..0] OF FSSpec;
   FSSpecArrayPtr    =  ^FSSpecArray;
   FSSpecArrayHandle =  ^FSSpecArrayPtr;
```

File and Directory Parameter Blocks

```
TYPE
   ParamBlkType      =    (ioParam, fileParam, volumeParam, cntrlParam,
                           slotDevParam, multiDevParam, accessParam,
                           objParam, copyParam, wdParam, fidParam, csParam,
                           foreignPrivsParam);
```

```
ParmBlkPtr        =  ^ParamBlockRec;
ParamBlockRec     =                 {basic File Manager parameter block}
RECORD
   qLink:            QElemPtr;      {next queue entry}
   qType:            Integer;       {queue type}
   ioTrap:           Integer;       {routine trap}
   ioCmdAddr:        Ptr;           {routine address}
   ioCompletion:     ProcPtr;       {pointer to completion routine}
   ioResult:         OSErr;         {result code}
   ioNamePtr:        StringPtr;     {pointer to pathname}
   ioVRefNum:        Integer;       {volume specification}
CASE ParamBlkType OF
ioParam:
  (ioRefNum:         Integer;       {file reference number}
   ioVersNum:        SignedByte;    {version number}
   ioPermssn:        SignedByte;    {read/write permission}
   ioMisc:           Ptr;           {miscellaneous}
   ioBuffer:         Ptr;           {data buffer}
   ioReqCount:       LongInt;       {requested number of bytes}
   ioActCount:       LongInt;       {actual number of bytes}
   ioPosMode:        Integer;       {positioning mode and newline char.}
   ioPosOffset:      LongInt);      {positioning offset}
fileParam:
  (ioFRefNum:        Integer;       {file reference number}
   ioFVersNum:       SignedByte;    {file version number (unused)}
   filler1:          SignedByte;    {reserved}
   ioFDirIndex:      Integer;       {directory index}
   ioFlAttrib:       SignedByte;    {file attributes}
   ioFlVersNum:      SignedByte;    {file version number (unused)}
   ioFlFndrInfo:     FInfo;         {information used by the Finder}
   ioFlNum:          LongInt;       {file ID}
   ioFlStBlk:        Integer;       {first alloc. blk. of data fork}
   ioFlLgLen:        LongInt;       {logical EOF of data fork}
   ioFlPyLen:        LongInt;       {physical EOF of data fork}
   ioFlRStBlk:       Integer;       {first alloc. blk. of resource fork}
   ioFlRLgLen:       LongInt;       {logical EOF of resource fork}
   ioFlRPyLen:       LongInt;       {physical EOF of resource fork}
   ioFlCrDat:        LongInt;       {date and time of creation}
   ioFlMdDat:        LongInt);      {date and time of last modification}
volumeParam:
  (filler2:          LongInt;       {reserved}
   ioVolIndex:       Integer;       {volume index}
   ioVCrDate:        LongInt;       {date and time of initialization}
```

```
    ioVLsBkUp:        LongInt;       {date and time of last modification}
    ioVAtrb:          Integer;       {volume attributes}
    ioVNmFls:         Integer;       {number of files in root directory}
    ioVDirSt:         Integer;       {first block of directory}
    ioVBlLn:          Integer;       {length of directory in blocks}
    ioVNmAlBlks:      Integer;       {number of allocation blocks}
    ioVAlBlkSiz:      LongInt;       {size of allocation blocks}
    ioVClpSiz:        LongInt;       {default clump size}
    ioAlBlSt:         Integer;       {first block in block map}
    ioVNxtFNum:       LongInt;       {next unused file ID}
    ioVFrBlk:         Integer);      {number of unused allocation blocks}
END;

HParmBlkPtr       =  ^HParamBlockRec;
HParamBlockRec    =                {HFS parameter block}
RECORD
    qLink:            QElemPtr;      {next queue entry}
    qType:            Integer;       {queue type}
    ioTrap:           Integer;       {routine trap}
    ioCmdAddr:        Ptr;           {routine address}
    ioCompletion:     ProcPtr;       {pointer to completion routine}
    ioResult:         OSErr;         {result code}
    ioNamePtr:        StringPtr;     {pointer to pathname}
    ioVRefNum:        Integer;       {volume specification}
CASE ParamBlkType OF
ioParam:
  (ioRefNum:          Integer;       {file reference number}
    ioVersNum:        SignedByte;    {version number}
    ioPermssn:        SignedByte;    {read/write permission}
    ioMisc:           Ptr;           {miscellaneous}
    ioBuffer:         Ptr;           {data buffer}
    ioReqCount:       LongInt;       {requested number of bytes}
    ioActCount:       LongInt;       {actual number of bytes}
    ioPosMode:        Integer;       {positioning mode and newline char.}
    ioPosOffset:      LongInt);      {positioning offset}
fileParam:
  (ioFRefNum:         Integer;       {file reference number}
    ioFVersNum:       SignedByte;    {file version number (unused)}
    filler1:          SignedByte;    {reserved}
    ioFDirIndex:      Integer;       {directory index}
    ioFlAttrib:       SignedByte;    {file attributes}
    ioFlVersNum:      SignedByte;    {file version number (unused)}
    ioFlFndrInfo:     FInfo;         {information used by the Finder}
    ioDirID:          LongInt;       {directory ID or file ID}
```

```
    ioFlStBlk:        Integer;       {first alloc. blk. of data fork}
    ioFlLgLen:        LongInt;       {logical EOF of data fork}
    ioFlPyLen:        LongInt;       {physical EOF of data fork}
    ioFlRStBlk:       Integer;       {first alloc. blk. of resource fork}
    ioFlRLgLen:       LongInt;       {logical EOF of resource fork}
    ioFlRPyLen:       LongInt;       {physical EOF of resource fork}
    ioFlCrDat:        LongInt;       {date and time of creation}
    ioFlMdDat:        LongInt);      {date and time of last modification}
volumeParam:
  (filler2:           LongInt;       {reserved}
    ioVolIndex:       Integer;       {volume index}
    ioVCrDate:        LongInt;       {date and time of initialization}
    ioVLsMod:         LongInt;       {date and time of last modification}
    ioVAtrb:          Integer;       {volume attributes}
    ioVNmFls:         Integer;       {number of files in root directory}
    ioVBitMap:        Integer;       {first block of volume bitmap}
    ioAllocPtr:       Integer;       {first block of next new file}
    ioVNmAlBlks:      Integer;       {number of allocation blocks}
    ioVAlBlkSiz:      LongInt;       {size of allocation blocks}
    ioVClpSiz:        LongInt;       {default clump size}
    ioAlBlSt:         Integer;       {first block in volume map}
    ioVNxtCNID:       LongInt;       {next unused node ID}
    ioVFrBlk:         Integer;       {number of unused allocation blocks}
    ioVSigWord:       Integer;       {volume signature}
    ioVDrvInfo:       Integer;       {drive number}
    ioVDRefNum:       Integer;       {driver reference number}
    ioVFSID:          Integer;       {file-system identifier}
    ioVBkUp:          LongInt;       {date and time of last backup}
    ioVSeqNum:        Integer;       {used internally}
    ioVWrCnt:         LongInt;       {volume write count}
    ioVFilCnt:        LongInt;       {number of files on volume}
    ioVDirCnt:        LongInt;       {number of directories on volume}
    ioVFndrInfo:      ARRAY[1..8] OF LongInt);
                                     {information used by the Finder}
accessParam:
  (filler3:           Integer;       {reserved}
    ioDenyModes:      Integer;       {access mode information}
    filler4:          Integer;       {reserved}
    filler5:          SignedByte;    {reserved}
    ioACUser:         SignedByte;    {user access rights}
    filler6:          LongInt;       {reserved}
    ioACOwnerID:      LongInt;       {owner ID}
    ioACGroupID:      LongInt;       {group ID}
    ioACAccess:       LongInt);      {directory access rights}
```

```
objParam:
  (filler7:          Integer;       {reserved}
   ioObjType:        Integer;       {function code}
   ioObjNamePtr:     Ptr;           {ptr to returned creator/group name}
   ioObjID:          LongInt);      {creator/group ID}
copyParam:
  (ioDstVRefNum:     Integer;       {destination volume identifier}
   filler8:          Integer;       {reserved}
   ioNewName:        Ptr;           {pointer to destination pathname}
   ioCopyName:       Ptr;           {pointer to optional name}
   ioNewDirID:       LongInt);      {destination directory ID}
wdParam:
  (filler9:          Integer;       {reserved}
   ioWDIndex:        Integer;       {working directory index}
   ioWDProcID:       LongInt;       {working directory user identifier}
   ioWDVRefNum:      Integer;       {working directory's vol. ref. num.}
   filler10:         Integer;       {reserved}
   filler11:         LongInt;       {reserved}
   filler12:         LongInt;       {reserved}
   filler13:         LongInt;       {reserved}
   ioWDDirID:        LongInt);      {working directory's directory ID}
fidParam:
  (filler14:         LongInt;       {reserved}
   ioDestNamePtr:    StringPtr;     {pointer to destination filename}
   filler15:         LongInt;       {reserved}
   ioDestDirID:      LongInt;       {destination parent directory ID}
   filler16:         LongInt;       {reserved}
   filler17:         LongInt;       {reserved}
   ioSrcDirID:       LongInt;       {source parent directory ID}
   filler18:         Integer;       {reserved}
   ioFileID:         LongInt);      {file ID}
csParam:
  (ioMatchPtr:       FSSpecArrayPtr;{pointer to array of matches}
   ioReqMatchCount:  LongInt;       {max. number of matches to return}
   ioActMatchCount:  LongInt;       {actual number of matches}
   ioSearchBits:     LongInt;       {enable bits for matching rules}
   ioSearchInfo1:    CInfoPBPtr;    {pointer to values and lower bounds}
   ioSearchInfo2:    CInfoPBPtr;    {pointer to masks and upper bounds}
   ioSearchTime:     LongInt;       {maximum time to search}
   ioCatPosition:    CatPositionRec;{current catalog position}
   ioOptBuffer:      Ptr;           {pointer to optional read buffer}
   ioOptBufSize:     LongInt);      {length of optional read buffer}
```

```
   foreignPrivParam:
     (filler21:              LongInt;    {reserved}
      filler22:              LongInt;    {reserved}
      ioForeignPrivBuffer:   Ptr;        {privileges data buffer}
      ioForeignPrivReqCount: LongInt;    {size of buffer}
      ioForeignPrivActCount: LongInt;    {amount of buffer used}
      filler23:              LongInt;    {reserved}
      ioForeignPrivDirID:    LongInt;    {parent directory ID of }
                                         { foreign file or directory}
      ioForeignPrivInfo1:    LongInt;    {privileges data}
      ioForeignPrivInfo2:    LongInt;    {privileges data}
      ioForeignPrivInfo3:    LongInt;    {privileges data}
      ioForeignPrivInfo4:    LongInt);   {privileges data}
   END;
```


Catalog Information Parameter Blocks

```
TYPE
   CInfoType         =  (hfileInfo, dirInfo);

   CInfoPBPtr        = ^CInfoPBRec;
   CInfoPBRec        =  {catalog information parameter block}
   RECORD
      qLink:            QElemPtr;      {next queue entry}
      qType:            Integer;       {queue type}
      ioTrap:           Integer;       {routine trap}
      ioCmdAddr:        Ptr;           {routine address}
      ioCompletion:     ProcPtr;       {pointer to completion routine}
      ioResult:         OSErr;         {result code}
      ioNamePtr:        StringPtr;     {pointer to pathname}
      ioVRefNum:        Integer;       {volume specification}
      ioFRefNum:        Integer;       {file reference number}
      ioFVersNum:       SignedByte;    {version number}
      filler1:          SignedByte;    {reserved}
      ioFDirIndex:      Integer;       {directory index}
      ioFlAttrib:       SignedByte;    {file or directory attributes}
      ioACUser:         SignedByte;    {directory access rights}
   CASE CInfoType OF
   hFileInfo:
     (ioFlFndrInfo:     FInfo;         {information used by the Finder}
      ioDirID:          LongInt;       {directory ID or file ID}
      ioFlStBlk:        Integer;       {first alloc. blk. of data fork}
      ioFlLgLen:        LongInt;       {logical EOF of data fork}
      ioFlPyLen:        LongInt;       {physical EOF of data fork}
```

```
      ioFlRStBlk:       Integer;       {first alloc. blk. of resource fork}
      ioFlRLgLen:       LongInt;       {logical EOF of resource fork}
      ioFlRPyLen:       LongInt;       {physical EOF of resource fork}
      ioFlCrDat:        LongInt;       {date and time of creation}
      ioFlMdDat:        LongInt;       {date and time of last modification}
      ioFlBkDat:        LongInt;       {date and time of last backup}
      ioFlXFndrInfo:    FXInfo;        {additional Finder information}
      ioFlParID:        LongInt;       {file parent directory ID}
      ioFlClpSiz:       LongInt);      {file's clump size}
   dirInfo:
     (ioDrUsrWds:       DInfo;         {information used by the Finder}
      ioDrDirID:        LongInt;       {directory ID}
      ioDrNmFls:        Integer;       {number of files in directory}
      filler3:          ARRAY[1..9] OF Integer;
      ioDrCrDat:        LongInt;       {date and time of creation}
      ioDrMdDat:        LongInt;       {date and time of last modification}
      ioDrBkDat:        LongInt;       {date and time of last backup}
      ioDrFndrInfo:     DXInfo;        {additional Finder information}
      ioDrParID:        LongInt);      {directory's parent directory ID}
   END;
```

Catalog Position Record

```
TYPE
   CatPositionRec    =           {catalog position record}
   RECORD
      initialize:       LongInt;                     {starting point}
      priv:             ARRAY[1..6] OF Integer;      {private data}
   END;
```

Catalog Move Parameter Block

```
TYPE
   CMovePBPtr        = ^CMovePBRec;
   CMovePBRec        =               {catalog move parameter block}
   RECORD
      qLink:            QElemPtr;     {next queue entry}
      qType:            Integer;      {queue type}
      ioTrap:           Integer;      {routine trap}
      ioCmdAddr:        Ptr;          {routine address}
      ioCompletion:     ProcPtr;      {pointer to completion routine}
      ioResult:         OSErr;        {result code}
      ioNamePtr:        StringPtr;    {pointer to pathname}
      ioVRefNum:        Integer;      {volume specification}
```

```
      filler1:          LongInt;       {reserved}
      ioNewName:        StringPtr;     {name of new directory}
      filler2:          LongInt;       {reserved}
      ioNewDirID:       LongInt;       {directory ID of new directory}
      filler3:          ARRAY[1..2] OF LongInt; {reserved}
      ioDirID:          LongInt;       {directory ID of current directory}
   END;
```


Working Directory Parameter Block

```
TYPE
   WDPBPtr           = ^WDPBRec;
   WDPBRec           =                {working directory parameter block}
   RECORD
      qLink:            QElemPtr;     {next queue entry}
      qType:            Integer;      {queue type}
      ioTrap:           Integer;      {routine trap}
      ioCmdAddr:        Ptr;          {routine address}
      ioCompletion:     ProcPtr;      {pointer to completion routine}
      ioResult:         OSErr;        {result code}
      ioNamePtr:        StringPtr;    {pointer to pathname}
      ioVRefNum:        Integer;      {volume specification}
      filler1:          Integer;      {reserved}
      ioWDIndex:        Integer;      {working directory index}
      ioWDProcID:       LongInt;      {working directory user identifier}
      ioWDVRefNum:      Integer;      {working directory's vol. ref. num.}
      filler2:          ARRAY[1..7] OF Integer; {reserved}
      ioWDDirID:        LongInt;      {working directory's directory ID}
   END;
```

File Control Block Parameter Block

```
TYPE
   FCBPBPtr          = ^FCBPBRec;
   FCBPBRec          =                {file control block parameter block}
   RECORD
      qLink:            QElemPtr;     {next queue entry}
      qType:            Integer;      {queue type}
      ioTrap:           Integer;      {routine trap}
      ioCmdAddr:        Ptr;          {routine address}
      ioCompletion:     ProcPtr;      {pointer to completion routine}
      ioResult:         OSErr;        {result code}
      ioNamePtr:        StringPtr;    {pointer to pathname}
      ioVRefNum:        Integer;      {volume specification}
```

```
      ioRefNum:         Integer;       {file reference number}
      filler:           Integer;       {reserved}
      ioFCBIndx:        Integer;       {FCB index}
      filler1:          Integer;       {reserved}
      ioFCBFlNm:        LongInt;       {file ID}
      ioFCBFlags:       Integer;       {flags}
      ioFCBStBlk:       Integer;       {first allocation block of file}
      ioFCBEOF:         LongInt;       {logical end-of-file}
      ioFCBPLen:        LongInt;       {physical end-of-file}
      ioFCBCrPs:        LongInt;       {position of the file mark}
      ioFCBVRefNum:     Integer;       {volume reference number}
      ioFCBClpSiz:      LongInt;       {file's clump size}
      ioFCBParID:       LongInt;       {parent directory ID}
   END;
```

Volume Attributes Buffer

```
TYPE
   GetVolParmsInfoBuffer =
   RECORD
      vMVersion:        Integer;       {version number}
      vMAttrib:         LongInt;       {volume attributes}
      vMLocalHand:      Handle;        {reserved}
      vMServerAdr:      LongInt;       {network server address}
      vMVolumeGrade:    LongInt;       {relative speed rating}
      vMForeignPrivID:  Integer;       {foreign privilege model}
   END;
```

Volume Mounting Information Records

```
TYPE
   VolumeType           =  OSType;
   VolMountInfoPtr      =  ^VolMountInfoHeader;
   VolMountInfoHeader   =              {volume mounting information}
   RECORD
      length:           Integer;       {length of mounting information}
      media:            VolumeType;    {type of volume}
   END;

   AFPVolMountInfoPtr   =  ^AFPVolMountInfo;
   AFPVolMountInfo      =              {AFP volume mounting information}
   RECORD
      length:           Integer;       {length of mounting information}
      media:            VolumeType;    {type of volume}
```

```
      flags:            Integer;       {reserved; must be set to 0}
      nbpInterval:      SignedByte;    {NBP retry interval}
      nbpCount:         SignedByte;    {NBP retry count}
      uamType:          Integer;       {user authentication method}
      zoneNameOffset:   Integer;       {offset to zone name}
      serverNameOffset: Integer;       {offset server name}
      volNameOffset:    Integer;       {offset to volume name}
      userNameOffset:   Integer;       {offset to user name}
      userPasswordOffset:
                        Integer;       {offset to user password}
      volPasswordOffset:
                        Integer;       {offset to volume password}
      AFPData:          PACKED ARRAY[1..144] OF CHAR;
                                       {standard AFP mounting info}
   END;
```


Internal Data Types

Volume and File Control Blocks

```
TYPE
   VCB                  =           {volume control block}
   RECORD
      qLink:            QElemPtr;   {next queue entry}
      qType:            Integer;    {queue type}
      vcbFlags:         Integer;    {volume flags (bit 15 = 1 if dirty)}
      vcbSigWord:       Integer;    {volume signature}
      vcbCrDate:        LongInt;    {date and time of volume creation}
      vcbLsMod:         LongInt;    {date and time of last modification}
      vcbAtrb:          Integer;    {volume attributes}
      vcbNmFls:         Integer;    {number of files in root directory}
      vcbVBMSt:         Integer;    {first block of volume bitmap}
      vcbAllocPtr:      Integer;    {start of next allocation search}
      vcbNmAlBlks:      Integer;    {number of allocation blocks in volume}
      vcbAlBlkSiz:      LongInt;    {size (in bytes) of allocation blocks}
      vcbClpSiz:        LongInt;    {default clump size}
      vcbAlBlSt:        Integer;    {first allocation block in volume}
      vcbNxtCNID:       LongInt;    {next unused catalog node ID}
      vcbFreeBks:       Integer;    {number of unused allocation blocks}
      vcbVN:            String[27]; {volume name}
      vcbDrvNum:        Integer;    {drive number}
      vcbDRefNum:       Integer;    {driver reference number}
      vcbFSID:          Integer;    {file-system identifier}
```

```
   vcbVRefNum:       Integer;    {volume reference number}
   vcbMAdr:          Ptr;        {used internally}
   vcbBufAdr:        Ptr;        {used internally}
   vcbMLen:          Integer;    {used internally}
   vcbDirIndex:      Integer;    {used internally}
   vcbDirBlk:        Integer;    {used internally}
   vcbVolBkUp:       LongInt;    {date and time of last backup}
   vcbVSeqNum:       Integer;    {volume backup sequence number}
   vcbWrCnt:         LongInt;    {volume write count}
   vcbXTClpSiz:      LongInt;    {clump size for extents overflow file}
   vcbCTClpSiz:      LongInt;    {clump size for catalog file}
   vcbNmRtDirs:      Integer;    {number of directories in root dir.}
   vcbFilCnt:        LongInt;    {number of files in volume}
   vcbDirCnt:        LongInt;    {number of directories in volume}
   vcbFndrInfo:      ARRAY[1..8] OF LongInt;
                                 {information used by the Finder}
   vcbVCSize:        Integer;    {used internally}
   vcbVBMCSiz:       Integer;    {used internally}
   vcbCtlCSiz:       Integer;    {used internally}
   vcbXTAlBlks:      Integer;    {size of extents overflow file}
   vcbCTAlBlks:      Integer;    {size of catalog file}
   vcbXTRef:         Integer;    {ref. num. for extents overflow file}
   vcbCTRef:         Integer;    {ref. num. for catalog file}
   vcbCtlBuf:        Ptr;        {ptr. to extents and catalog caches}
   vcbDirIDM:        LongInt;    {directory last searched}
   vcbOffsM:         Integer;    {offspring index at last search}
END;

FCB                  =           {file control block}
RECORD
   fcbFlNum:         LongInt;    {file ID}
   fcbFlags:         Integer;    {file flags}
   fcbSBlk:          Integer;    {first allocation block of file}
   fcbEOF:           LongInt;    {logical end-of-file}
   fcbPLen:          LongInt;    {physical end-of-file}
   fcbCrPs:          LongInt;    {current file mark position}
   fcbVPtr:          Ptr;        {pointer to volume control block}
   fcbBfAdr:         Ptr;        {pointer to access path buffer}
   fcbFlPos:         Integer;    {reserved}
   fcbClmpSize:      LongInt;    {file clump size}
   fcbBTCBPtr:       Ptr;        {pointer to B*-tree control block}
   fcbExtRec:        ExtDataRec; {first three file extents}
   fcbFType:         LongInt;    {file's four Finder type bytes}
```

```
      fcbCatPos:        LongInt;    {catalog hint for use on Close}
      fcbDirID:         LongInt;    {file's parent directory ID}
      fcbCName:         String[31]; {name of file}
   END;
```

Drive Queue Elements

```
TYPE
   DrvQEl               =             {drive queue element}
   RECORD
      qLink:            QElemPtr;     {next queue entry}
      qType:            Integer;      {flag for dQDrvSz and dQDrvSz2}
      dQDrive:          Integer;      {drive number}
      dQRefNum:         Integer;      {driver reference number}
      dQFSID:           Integer;      {file-system identifier}
      dQDrvSz:          Integer;      {number of logical blocks on drive}
      dQDrvSz2:         Integer;      {additional field for large drives}
   END;
```

High-Level File Access Routines

Reading, Writing, and Closing Files

```
FUNCTION FSRead             (refNum: Integer; VAR count: LongInt;
                             buffPtr: Ptr): OSErr;
FUNCTION FSWrite            (refNum: Integer; VAR count: LongInt;
                             buffPtr: Ptr): OSErr;
FUNCTION FSClose            (refNum: Integer): OSErr;
```

Manipulating the File Mark

```
FUNCTION GetFPos            (refNum: Integer; VAR filePos: LongInt): OSErr;
FUNCTION SetFPos            (refNum: Integer; posMode: Integer;
                             posOff: LongInt): OSErr;
```

Manipulating the End-of-File

```
FUNCTION GetEOF             (refNum: Integer; VAR logEOF: LongInt): OSErr;
FUNCTION SetEOF             (refNum: Integer; logEOF: LongInt): OSErr;
```

Allocating File Blocks

```
FUNCTION Allocate           (refNum: Integer; VAR count: LongInt): OSErr;
FUNCTION AllocContig        (refNum: Integer; VAR count: LongInt): OSErr;
```

Low-Level File Access Routines

Reading, Writing, and Closing Files

```
FUNCTION PBRead             (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBReadSync         (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBReadAsync        (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBWrite            (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBWriteSync        (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBWriteAsync       (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBClose            (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBCloseSync        (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBCloseAsync       (paramBlock: ParmBlkPtr): OSErr;
```

Manipulating the File Mark

```
FUNCTION PBGetFPos          (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBGetFPosSync      (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBGetFPosAsync     (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBSetFPos          (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBSetFPosSync      (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBSetFPosAsync     (paramBlock: ParmBlkPtr): OSErr;
```

Manipulating the End-of-File

```
FUNCTION PBGetEOF           (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBGetEOFSync       (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBGetEOFAsync      (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBSetEOF           (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBSetEOFSync       (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBSetEOFAsync      (paramBlock: ParmBlkPtr): OSErr;
```

Allocating File Blocks

```
FUNCTION PBAllocate         (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBAllocateSync     (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBAllocateAsync    (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBAllocContig      (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBAllocContigSync  (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBAllocContigAsync (paramBlock: ParmBlkPtr): OSErr;
```

Updating Files

```
FUNCTION PBFlushFile        (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBFlushFileSync    (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBFlushFileAsync   (paramBlock: ParmBlkPtr): OSErr;
```

High-Level Volume Access Routines

Unmounting Volumes

```
FUNCTION UnmountVol         (volName: StringPtr; vRefNum: Integer): OSErr;
FUNCTION Eject              (volName: StringPtr; vRefNum: Integer): OSErr;
```

Updating Volumes

```
FUNCTION FlushVol           (volName: StringPtr; vRefNum: Integer): OSErr;
```

Manipulating the Default Volume

```
FUNCTION GetVol             (volName: StringPtr; VAR vRefNum: Integer):
                             OSErr;
FUNCTION SetVol             (volName: StringPtr; vRefNum: Integer): OSErr;
FUNCTION HGetVol            (volName: StringPtr; VAR vRefNum: Integer;
                             VAR dirID: LongInt): OSErr;
FUNCTION HSetVol            (volName: StringPtr; vRefNum: Integer;
                             dirID: LongInt): OSErr;
```

Obtaining Volume Information

```
FUNCTION GetVInfo           (drvNum: Integer; volName: StringPtr;
                             VAR vRefNum: Integer; VAR freeBytes: LongInt):
                             OSErr;
FUNCTION GetVRefNum         (refNum: Integer; VAR vRefNum: Integer): OSErr;
```

Low-Level Volume Access Routines

Mounting and Unmounting Volumes

```
FUNCTION PBMountVol         (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBUnmountVol       (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBEject            (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBOffLine          (paramBlock: ParmBlkPtr): OSErr;
```

Updating Volumes

```
FUNCTION PBFlushVol          (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBFlushVolSync      (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBFlushVolAsync     (paramBlock: ParmBlkPtr): OSErr;
```

Obtaining Volume Information

```
FUNCTION PBHGetVInfo         (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBHGetVInfoSync     (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHGetVInfoAsync    (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBSetVInfo          (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBSetVInfoSync      (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBSetVInfoAsync     (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHGetVolParms      (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBHGetVolParmsSync  (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHGetVolParmsAsync (paramBlock: HParmBlkPtr): OSErr;
```

Manipulating the Default Volume

```
FUNCTION PBGetVol            (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBGetVolSync        (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBGetVolAsync       (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBSetVol            (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBSetVolSync        (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBSetVolAsync       (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBHGetVol           (paramBlock: WDPBPtr; async: Boolean): OSErr;
FUNCTION PBHGetVolSync       (paramBlock: WDPBPtr): OSErr;
FUNCTION PBHGetVolAsync      (paramBlock: WDPBPtr): OSErr;
FUNCTION PBHSetVol           (paramBlock: WDPBPtr; async: Boolean): OSErr;
FUNCTION PBHSetVolSync       (paramBlock: WDPBPtr): OSErr;
FUNCTION PBHSetVolAsync      (paramBlock: WDPBPtr): OSErr;
```

File System Specification Routines

Opening Files

```
FUNCTION FSpOpenDF           (spec: FSSpec; permission: SignedByte;
                              VAR refNum: Integer): OSErr;
FUNCTION FSpOpenRF           (spec: FSSpec; permission: SignedByte;
                              VAR refNum: Integer): OSErr;
```

Creating and Deleting Files and Directories

```
FUNCTION FSpCreate          (spec: FSSpec; creator: OSType;
                             fileType: OSType; scriptTag: ScriptCode):
                             OSErr;
FUNCTION FSpDirCreate       (spec: FSSpec; scriptTag: ScriptCode;
                             VAR createdDirID: LongInt): OSErr;
FUNCTION FSpDelete          (spec: FSSpec): OSErr;
```

Accessing Information About Files and Directories

```
FUNCTION FSpGetFInfo        (spec: FSSpec; VAR fndrInfo: FInfo): OSErr;
FUNCTION FSpSetFInfo        (spec: FSSpec; fndrInfo: FInfo): OSErr;
FUNCTION FSpSetFLock        (spec: FSSpec): OSErr;
FUNCTION FSpRstFLock        (spec: FSSpec): OSErr;
FUNCTION FSpRename          (spec: FSSpec; newName: Str255): OSErr;
```

Moving Files or Directories

```
FUNCTION FSpCatMove         (source: FSSpec; dest: FSSpec): OSErr;
```

Exchanging the Data in Two Files

```
FUNCTION FSpExchangeFiles   (source: FSSpec; dest: FSSpec): OSErr;
```

Creating File System Specifications

```
FUNCTION FSMakeFSSpec       (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255; VAR spec: FSSpec): OSErr;
FUNCTION PBMakeFSSpec       (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBMakeFSSpecSync   (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBMakeFSSpecAsync  (paramBlock: HParmBlkPtr): OSErr;
```

High-Level HFS Routines

Opening Files

```
FUNCTION HOpenDF            (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255; permission: SignedByte;
                             VAR refNum: Integer): OSErr;
FUNCTION HOpenRF            (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255; permission: SignedByte;
                             VAR refNum: Integer): OSErr;
```

```
FUNCTION HOpen              (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255; permission: SignedByte;
                             VAR refNum: Integer): OSErr;
```

Creating and Deleting Files and Directories

```
FUNCTION HCreate            (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255; creator: OSType;
                             fileType: OSType): OSErr;
FUNCTION DirCreate          (vRefNum: Integer; parentDirID: LongInt;
                             directoryName: Str255;
                             VAR createdDirID: LongInt): OSErr;
FUNCTION HDelete            (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255): OSErr;
```

Accessing Information About Files and Directories

```
FUNCTION HGetFInfo          (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255; VAR fndrInfo: FInfo): OSErr;
FUNCTION HSetFInfo          (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255; fndrInfo: FInfo): OSErr;
FUNCTION HSetFLock          (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255): OSErr;
FUNCTION HRstFLock          (vRefNum: Integer; dirID: LongInt;
                             fileName: Str255): OSErr;
FUNCTION HRename            (vRefNum: Integer; dirID: LongInt;
                             oldName: Str255; newName: Str255): OSErr;
```

Moving Files or Directories

```
FUNCTION CatMove            (vRefNum: Integer; dirID: LongInt;
                             oldName: Str255; newDirID: LongInt;
                             newName: Str255): OSErr;
```

Maintaining Working Directories

```
FUNCTION OpenWD             (vRefNum: Integer; dirID: LongInt;
                             procID: LongInt; VAR wdRefNum: Integer): OSErr;
FUNCTION CloseWD            (wdRefNum: Integer): OSErr;
FUNCTION GetWDInfo          (wdRefNum: Integer; VAR vRefNum: Integer;
                             VAR dirID: LongInt; VAR procID: LongInt):
                             OSErr;
```

Low-Level HFS Routines

Opening Files

```
FUNCTION PBHOpenDF          (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBHOpenDFSync      (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHOpenDFAsync     (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHOpenRF          (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBHOpenRFSync      (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHOpenRFAsync     (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHOpen            (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBHOpenSync        (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHOpenAsync       (paramBlock: HParmBlkPtr): OSErr;
```

Creating and Deleting Files and Directories

```
FUNCTION PBHCreate          (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBHCreateSync      (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHCreateAsync     (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBDirCreate        (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBDirCreateSync    (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBDirCreateAsync   (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHDelete          (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBHDeleteSync      (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHDeleteAsync     (paramBlock: HParmBlkPtr): OSErr;
```

Accessing Information About Files and Directories

```
FUNCTION PBGetCatInfo       (paramBlock: CInfoPBPtr; async: Boolean): OSErr;
FUNCTION PBGetCatInfoSync   (paramBlock: CInfoPBPtr): OSErr;
FUNCTION PBGetCatInfoAsync  (paramBlock: CInfoPBPtr): OSErr;
FUNCTION PBSetCatInfo       (paramBlock: CInfoPBPtr; async: Boolean): OSErr;
FUNCTION PBSetCatInfoSync   (paramBlock: CInfoPBPtr): OSErr;
FUNCTION PBSetCatInfoAsync  (paramBlock: CInfoPBPtr): OSErr;
FUNCTION PBHGetFInfo        (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
```

```
FUNCTION PBHGetFInfoSync    (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHGetFInfoAsync   (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHSetFInfo        (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBHSetFInfoSync    (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHSetFInfoAsync   (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHSetFLock        (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBHSetFLockSync    (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHSetFLockAsync   (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHRstFLock        (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBHRstFLockSync    (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHRstFLockAsync   (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHRename          (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBHRenameSync      (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHRenameAsync     (paramBlock: HParmBlkPtr): OSErr;
```

Moving Files or Directories

```
FUNCTION PBCatMove          (paramBlock: CMovePBPtr; async: Boolean): OSErr;
FUNCTION PBCatMoveSync      (paramBlock: CMovePBPtr): OSErr;
FUNCTION PBCatMoveAsync     (paramBlock: CMovePBPtr): OSErr;
```

Maintaining Working Directories

```
FUNCTION PBOpenWD           (paramBlock: WDPBPtr; async: Boolean): OSErr;
FUNCTION PBOpenWDSync       (paramBlock: WDPBPtr): OSErr;
FUNCTION PBOpenWDAsync      (paramBlock: WDPBPtr): OSErr;
FUNCTION PBCloseWD          (paramBlock: WDPBPtr; async: Boolean): OSErr;
FUNCTION PBCloseWDSync      (paramBlock: WDPBPtr): OSErr;
FUNCTION PBCloseWDAsync     (paramBlock: WDPBPtr): OSErr;
FUNCTION PBGetWDInfo        (paramBlock: WDPBPtr; async: Boolean): OSErr;
FUNCTION PBGetWDInfoSync    (paramBlock: WDPBPtr): OSErr;
FUNCTION PBGetWDInfoAsync   (paramBlock: WDPBPtr): OSErr;
```

Searching a Catalog

```
FUNCTION PBCatSearch        (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBCatSearchSync    (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBCatSearchAsync   (paramBlock: HParmBlkPtr): OSErr;
```

Exchanging the Data in Two Files

```
FUNCTION PBExchangeFiles     (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBExchangeFilesSync (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBExchangeFilesAsync
                             (paramBlock: HParmBlkPtr): OSErr;
```

Shared Environment Routines

Opening Files While Denying Access

```
FUNCTION PBHOpenDeny         (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBHOpenDenySync     (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHOpenDenyAsync    (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHOpenRFDeny       (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBHOpenRFDenySync   (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHOpenRFDenyAsync  (paramBlock: HParmBlkPtr): OSErr;
```

Locking and Unlocking File Ranges

```
FUNCTION PBLockRange         (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBLockRangeSync     (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBLockRangeAsync    (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBUnlockRange       (paramBlock: ParmBlkPtr; async: Boolean): OSErr;
FUNCTION PBUnlockRangeSync   (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBUnlockRangeAsync  (paramBlock: ParmBlkPtr): OSErr;
```

Manipulating Share Points

```
FUNCTION PBShare             (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBShareSync         (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBShareAsync        (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBUnshare           (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBUnshareSync       (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBUnshareAsync      (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBGetUGEntry        (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBGetUGEntrySync    (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBGetUGEntryAsync   (paramBlock: HParmBlkPtr): OSErr;
```

Controlling Directory Access

```
FUNCTION PBHGetDirAccess     (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBHGetDirAccessSync (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHGetDirAccessAsync
                             (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHSetDirAccess     (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBHSetDirAccessSync (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHSetDirAccessAsync
                             (paramBlock: HParmBlkPtr): OSErr;
```

Mounting Volumes

```
FUNCTION PBGetVolMountInfoSize
                             (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBGetVolMountInfo   (paramBlock: ParmBlkPtr): OSErr;
FUNCTION PBVolumeMount       (paramBlock: ParmBlkPtr): OSErr;
```

Controlling Login Access

```
FUNCTION PBHGetLogInInfo     (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBHGetLogInInfoSync (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHGetLogInInfoAsync
                             (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHMapID            (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBHMapIDSync        (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHMapIDAsync       (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHMapName          (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBHMapNameSync      (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHMapNameAsync     (paramBlock: HParmBlkPtr): OSErr;
```

Copying and Moving Files

```
FUNCTION PBHCopyFile         (paramBlock: HParmBlkPtr; async: Boolean):
                              OSErr;
FUNCTION PBHCopyFileSync     (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHCopyFileAsync    (paramBlock: HParmBlkPtr): OSErr;
```

```
FUNCTION PBHMoveRename      (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBHMoveRenameSync  (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBHMoveRenameAsync (paramBlock: HParmBlkPtr): OSErr;
```

File ID Routines

Resolving File ID References

```
FUNCTION PBResolveFileIDRef (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBResolveFileIDRefSync
                            (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBResolveFileIDRefAsync
                            (paramBlock: HParmBlkPtr): OSErr;
```

Creating and Deleting File ID References

```
FUNCTION PBCreateFileIDRef  (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBCreateFileIDRefSync
                            (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBCreateFileIDRefAsync
                            (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBDeleteFileIDRef  (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBDeleteFileIDRefSync
                            (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBDeleteFileIDRefAsync
                            (paramBlock: HParmBlkPtr): OSErr;
```

Foreign File System Routines

Accessing Privilege Information in Foreign File Systems

```
FUNCTION PBGetForeignPrivs  (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBGetForeignPrivsSync
                            (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBGetForeignPrivsAsync
                            (paramBlock: HParmBlkPtr): OSErr;
```

```
FUNCTION PBSetForeignPrivs  (paramBlock: HParmBlkPtr; async: Boolean):
                             OSErr;
FUNCTION PBSetForeignPrivsSync
                            (paramBlock: HParmBlkPtr): OSErr;
FUNCTION PBSetForeignPrivsAsync
                            (paramBlock: HParmBlkPtr): OSErr;
```

Utility Routines

Obtaining Queue Headers

```
FUNCTION GetFSQHdr          : QHdrPtr;
FUNCTION GetVCBQHdr         : QHdrPtr;
FUNCTION GetDrvQHdr         : QHdrPtr;
```

Adding a Drive

```
PROCEDURE AddDrive          (drvrRefNum: Integer; drvNum: Integer;
                             qEl: DrvQElPtr);
```

Obtaining File Control Block Information

```
FUNCTION PBGetFCBInfo       (paramBlock: FCBPBPtr; async: Boolean): OSErr;
FUNCTION PBGetFCBInfoSync   (paramBlock: FCBPBPtr): OSErr;
FUNCTION PBGetFCBInfoAsync  (paramBlock: FCBPBPtr): OSErr;
```

Application-Defined Routine

Completion Routines

```
PROCEDURE MyCompletionProc;
```

C Summary

Constants

```
/*Gestalt constants*/
#define gestaltFSAttr            'fs  '   /*file system attributes selector*/
#define gestaltFullExtFSDispatching 0     /*exports HFSDispatch traps*/
#define gestaltHasFSSpecCalls       1     /*supports FSSpec records*/
```

```
/*directory IDs*/
enum {
   fsRtParID            = 1,     /*directory ID of root directory's parent*/
   fsRtDirID            = 2};    /*directory ID of volume's root directory*/

/*values for requesting file read/write permissions*/
enum {
   fsCurPerm            = 0,     /*whatever permission is allowed*/
   fsRdPerm             = 1,     /*read permission*/
   fsWrPerm             = 2,     /*write permission*/
   fsRdWrPerm           = 3,     /*exclusive read/write permission*/
   fsRdWrShPerm         = 4};    /*shared read/write permission*/

/*file mark positioning modes*/
enum {
   fsAtMark             = 0,     /*at current mark}
   fsFromStart          = 1,     /*set mark relative to beginning of file*/
   fsFromLEOF           = 2,     /*set mark relative to logical end-of-file*/
   fsFromMark           = 3,     /*set mark relative to current mark*/
   rdVerify             = 64};   /*add to above for read-verify*/

/*values for ioSearchBits in PBCatSearch parameter block*/
enum {
   fsSBPartialName      = 1,        /*substring of name*/
   fsSBFullName         = 2,        /*full name*/
   fsSBFlAttrib         = 4,        /*directory flag; software lock flag*/
   fsSBNegate           = 16384};   /*reverse match status*/

/*for files only*/
enum {
   fsSBFlFndrInfo       = 8,        /*Finder file info*/
   fsSBFlLgLen          = 32,       /*logical length of data fork*/
   fsSBFlPyLen          = 64,       /*physical length of data fork*/
   fsSBFlRLgLen         = 128,      /*logical length of resource fork*/
   fsSBFlRPyLen         = 256,      /*physical length of resource fork*/
   fsSBFlCrDat          = 512,      /*file creation date*/
   fsSBFlMdDat          = 1024,     /*file modification date*/
   fsSBFlBkDat          = 2048,     /*file backup date*/
   fsSBFlXFndrInfo      = 4096,     /*more Finder file info*/
   fsSBFlParID          = 8192};    /*file's parent ID*/
```

```
/*for directories only*/
enum {
   fsSBDrUsrWds         = 8,        /*Finder directory info*/
   fsSBDrNmFls          = 16,       /*number of files in directory*/
   fsSBDrCrDat          = 512,      /*directory creation date*/
   fsSBDrMdDat          = 1024,     /*directory modification date*/
   fsSBDrBkDat          = 2048,     /*directory backup date*/
   fsSBDrFndrInfo       = 4096,     /*more Finder directory info*/
   fsSBDrParID          = 8192};    /*directory's parent ID*/

/*value of vMForeignPrivID in file attributes buffer*/
enum {fsUnixPriv        = 1};       /*A/UX privilege model*/

/*bit positions in vMAttrib field of GetVolParmsInfoBuffer*/
enum {
   bHasBlankAccessPrivileges
                        = 4,        /*volume supports inherited privileges*/
   bHasBTreeMgr         = 5,        /*reserved*/
   bHasFileIDs          = 6,        /*volume supports file ID functions*/
   bHasCatSearch        = 7,        /*volume supports PBCatSearch*/
   bHasUserGroupList    = 8,        /*volume supports AFP privileges*/
   bHasPersonalAccessPrivileges
                        = 9,        /*local file sharing is enabled*/
   bHasFolderLock       = 10,       /*volume supports locking of folders*/
   bHasShortName        = 11,       /*volume supports shorter volume name*/
   bHasDesktopMgr       = 12,       /*volume supports Desktop Manager*/
   bHasMoveRename       = 13,       /*volume supports _MoveRename*/
   bHasCopyFile         = 14,       /*volume supports _CopyFile*/
   bHasOpenDeny         = 15,       /*volume supports shared access modes*/
   bHasExtFSVol         = 16,       /*volume is external file system volume*/
   bNoSysDir            = 17,       /*volume has no system directory*/
   bAccessCntl          = 18,       /*volume supports AFP access control*/
   bNoBootBlks          = 19,       /*volume is not a startup volume*/
   bNoDeskItems         = 20,       /*do not place objects on the desktop*/
   bNoSwitchTo          = 25,       /*do not switch launch to applications*/
   bTrshOffLine         = 26,       /*zoom volume when it is unmounted*/
   bNoLclSync           = 27,       /*don't let Finder change mod. date*/
   bNoVNEdit            = 28,       /*lock volume name*/
   bNoMiniFndr          = 29,       /*reserved; always 1*/
   bLocalWList          = 30,       /*use shared volume handle for window */
                                    /* list*/
   bLimitFCBs           = 31};      /*limit file control blocks*/
```

```
/*media type in remote mounting information/*
enum {AppleShareMediaType
                        = 'afpm'};  /*an AppleShare volume*/

/*user authentication methods in AFP remote mounting information*/
enum {
   kNoUserAuthentication   = 1,        /*guest status; no password needed*/
   kPassword               = 2,        /*8-byte password*/
   kEncryptPassword        = 3,        /*encrypted 8-byte password*/
   kTwoWayEncryptPassword  = 6};       /*two-way random encryption; */
                                       /* authenticate both user and server*/
```


Data Types

File System Specification Record

```
struct FSSpec  {                    /*file system specification*/
      short       vRefNum;          /*volume reference number*/
      long        parID;            /*directory ID of parent directory*/
      Str63       name;             /*filename or directory name*/
};

typedef struct FSSpec FSSpec;
typedef FSSpec *FSSpecPtr;
typedef FSSpecPtr *FSSpecHandle;
```

File and Directory Parameter Blocks

```
union ParamBlockRec {
      IOParam           ioParam;
      FileParam         fileParam;
      VolumeParam       volumeParam;
      CntrlParam        cntrlParam;
      SlotDevParam      slotDevParam;
      MultiDevParam     multiDevParam;
};

typedef union ParamBlockRec ParamBlockRec;
typedef ParamBlockRec *ParmBlkPtr;
```

```
#define ParamBlockHeader \
      QElemPtr    qLink;           /*next queue entry*/\
      short       qType;           /*queue type*/\
      short       ioTrap;          /*routine trap*/\
      Ptr         ioCmdAddr;       /*routine address*/\
      ProcPtr     ioCompletion;    /*completion routine*/\
      OSErr       ioResult;        /*result code*/\
      StringPtr   ioNamePtr;       /*pointer to pathname*/\
      short       ioVRefNum;       /*volume specification*/

struct IOParam {
      ParamBlockHeader
      short       ioRefNum;        /*file reference number*/
      char        ioVersNum;       /*version number*/
      char        ioPermssn;       /*read/write permission*/
      Ptr         ioMmisc;         /*miscellaneous*/
      Ptr         ioBuffer;        /*data buffer*/
      long        ioReqCount;      /*requested number of bytes*/
      long        ioActCount;      /*actual number of bytes*/
      short       ioPosMode;       /*positioning mode and newline char.*/
      long        ioPosOffset;     /*positioning offset*/
};

typedef struct IOParam IOParam;

struct FileParam {
      ParamBlockHeader
      short          ioFRefNum;       /*file reference number*/
      char           ioFVersNum;      /*file version number (unused)*/
      char           filler1;         /*reserved*/
      short          ioFDirIndex;     /*directory index*/
      unsigned char  ioFlAttrib;      /*file attributes*/
      unsigned char  ioFlVersNum;     /*file version number (unused)*/
      FInfo          ioFlFndrInfo;    /*information used by the Finder*/
      unsigned long  ioFlNum;         /*File ID*/
      unsigned short ioFlStBlk;       /*first alloc. blk. of data fork*/
      long           ioFlLgLen;       /*logical EOF of data fork*/
      long           ioFlPyLen;       /*physical EOF of data fork*/
      unsigned short ioFlRStBlk;      /*first alloc. blk. of resource fork*/
      long           ioFlRLgLen;      /*logical EOF of resource fork*/
      long           ioFlRPyLen;      /*physical EOF of resource fork*/
      unsigned long  ioFlCrDat;       /*date and time of creation*/
      unsigned long  ioFlMdDat;       /*date and time of last modification*/
};

typedef struct FileParam FileParam;
```

```
struct VolumeParam {
      ParamBlockHeader
      long           filler2;         /*reserved*/
      short          ioVolIndex;      /*volume index*/
      unsigned long  ioVCrDate;       /*date and time of initialization*/
      unsigned long  ioVLsBkUp;       /*date and time of last modification*/
      unsigned short ioVAtrb;         /*volume attributes*/
      unsigned short ioVNmFls;        /*number of files in root directory*/
      unsigned short ioVDirSt;        /*first block of directory*/
      short          ioVBlLn;         /*length of directory in blocks*/
      unsigned short ioVNmAlBlks;     /*number of allocation blocks*/
      long           ioVAlBlkSiz;     /*size of allocation blocks*/
      long           ioVClpSiz;       /*number of bytes to allocate*/
      unsigned short ioAlBlSt;        /*first block in block map*/
      unsigned long  ioVNxtFNum;      /*next unused file ID*/
      unsigned short ioVFrBlk;        /*number of unused allocation blocks*/
};

typedef struct VolumeParam VolumeParam;

union HParamBlockRec {                /*HFS parameter block*/
      HIOParam            ioParam;
      HFileParam          fileParam;
      HVolumeParam        volumeParam;
      AccessParam         accessParam;
      ObjParam            objParam;
      CopyParam           copyParam;
      WDParam             wdParam;
      FIDParam            fidParam;
      CSParam             csParam;
      ForeignPrivParam    foreignPrivParam;
};

typedef union HParamBlockRec HParamBlockRec;
typedef HParamBlockRec *HParmBlkPtr;

struct HIOParam {
      ParamBlockHeader
      short            ioRefNum;      /*file reference number*/
      char             ioVersNum;     /*version number*/
      char             ioPermssn;     /*read/write permission*/
      Ptr              ioMisc;        /*miscellaneous*/
      Ptr              ioBuffer;      /*data buffer*/
      long             ioReqCount;    /*requested number of bytes*/
```

```
      long              ioActCount;    /*actual number of bytes*/
      short             ioPosMode;     /*positioning mode and newline char.*/
      long              ioPosOffset;   /*positioning offset*/
};

typedef struct HIOParam HIOParam;

struct HFileParam {
      ParamBlockHeader
      short             ioFRefNum;     /*file reference number*/
      char              ioFVersNum;    /*file version number (unused)*/
      char              filler1;       /*reserved*/
      short             ioFDirIndex;   /*directory index*/
      char              ioFlAttrib;    /*file attributes*/
      char              ioFlVersNum;   /*file version number (unused)*/
      FInfo             ioFlFndrInfo;  /*information used by the Finder*/
      long              ioDirID;       /*directory ID or file ID*/
      unsigned short    ioFlStBlk;     /*first alloc. blk. of data fork*/
      long              ioFlLgLen;     /*logical EOF of data fork*/
      long              ioFlPyLen;     /*physical EOF of data fork*/
      unsigned short    ioFlRStBlk;    /*first alloc. blk. of resource fork*/
      long              ioFlRLgLen;    /*logical EOF of resource fork*/
      long              ioFlRPyLen;    /*physical EOF of resource fork*/
      unsigned long     ioFlCrDat;     /*date and time of creation*/
      unsigned long     ioFlMdDat;     /*date and time of last modification*/
};

typedef struct HFileParam HFileParam;

struct HVolumeParam {
      ParamBlockHeader
      long              filler2;       /*reserved*/
      short             ioVolIndex;    /*volume index*/
      unsigned long     ioVCrDate;     /*date and time of initialization*/
      unsigned long     ioVLsMod;      /*date and time of last modification*/
      short             ioVAtrb;       /*volume attributes*/
      unsigned short    ioVNmFls;      /*number of files in root directory*/
      short             ioVBitMap;     /*first block of volume bitmap*/
      short             ioAllocPtr;    /*first block of next new file*/
      unsigned short    ioVNmAlBlks;   /*number of allocation blocks*/
      long              ioVAlBlkSiz;   /*size of allocation blocks*/
      long              ioVClpSiz;     /*default clump size*/
      short             ioAlBlSt;      /*first block in volume map*/
      long              ioVNxtCNID;    /*next unused node ID*/
```

```
      unsigned short    ioVFrBlk;      /*number of unused allocation blocks*/
      unsigned short    ioVSigWord;    /*volume signature*/
      short             ioVDrvInfo;    /*drive number*/
      short             ioVDRefNum;    /*driver reference number*/
      short             ioVFSID;       /*file-system identifier*/
      unsigned long     ioVBkUp;       /*date and time of last backup*/
      unsigned short    ioVSeqNum;     /*used internally*/
      long              ioVWrCnt;      /*volume write count*/
      long              ioVFilCnt;     /*number of files on volume*/
      long              ioVDirCnt;     /*number of directories on volume*/
      long              ioVFndrInfo[8];/*information used by the Finder*/
};

typedef struct HVolumeParam HVolumeParam;

struct AccessParam {
      ParamBlockHeader
      short             filler3;       /*reserved*/
      short             ioDenyModes;   /*access mode information*/
      short             filler4;       /*reserved*/
      char              filler5;       /*reserved*/
      char              ioACUser;      /*user access rights*/
      long              filler6;       /*reserved*/
      long              ioACOwnerID;   /*owner ID*/
      long              ioACGroupID;   /*group ID*/
      long              ioACAccess;    /*directory access rights*/
};

typedef struct AccessParam AccessParam;

struct ObjParam {
      ParamBlockHeader
      short             filler7;       /*reserved*/
      short             ioObjType;     /*function code*/
      StringPtr         ioObjNamePtr;  /*ptr to returned creator/group name*/
      long              ioObjID;       /*creator/group ID*/
      long              ioReqCount;    /*size of buffer area*/
      long              ioActCount;    /*length of data*/
};

typedef struct ObjParam ObjParam;
```

```
struct CopyParam {
      ParamBlockHeader
      short             ioDstVRefNum;  /*destination volume identifier*/
      short             filler8;       /*reserved*/
      StringPtr         ioNewName;     /*pointer to destination pathname*/
      StringPtr         ioCopyName;    /*pointer to optional name*/
      long              ioNewDirID;    /*destination directory ID*/
      long              filler14;      /*reserved*/
      long              filler15;      /*reserved*/
      long              ioDirID;       /*directory ID or file ID*/
};

typedef struct CopyParam CopyParam;

struct WDParam {
      ParamBlockHeader
      short             filler9;       /*reserved*/
      short             ioWDIndex;     /*working directory index*/
      long              ioWDProcID;    /*working directory user identifier*/
      short             ioWDVRefNum;   /*working directory's vol. ref. num.*/
      short             filler10;      /*reserved*/
      long              filler11;      /*reserved*/
      long              filler12;      /*reserved*/
      long              filler13;      /*reserved*/
      long              ioWDDirID;     /*working directory's directory ID*/
};

typedef struct WDParam WDParam;

struct FIDParam {
      ParamBlockHeader
      long              filler1;       /*reserved*/
      StringPtr         ioDestNamePtr; /*pointer to destination filename*/
      long              filler2;       /*reserved*/
      long              ioDestDirID;   /*destination parent directory ID*/
      long              filler3;       /*reserved*/
      long              filler4;       /*reserved*/
      long              ioSrcDirID;    /*source parent directory ID*/
      short             filler5;       /*reserved*/
      long              ioFileID;      /*file ID*/
};

typedef struct FIDParam FIDParam;
```

```
struct CSParam {
      ParamBlockHeader
      FSSpecPtr         ioMatchPtr;       /*pointer to array of matches*/
      long              ioReqMatchCount;  /*max number of matches to return*/
      long              ioActMatchCount;  /*actual number of matches*/
      long              ioSearchBits;     /*enable bits for matching rules*/
      CInfoPBPtr        ioSearchInfo1;    /*pointer to values and lower */
                                          /* bounds*/
      CInfoPBPtr        ioSearchInfo2;    /*pointer to masks and upper */
                                          /* bounds*/
      long              ioSearchTime;     /*maximum time to search*/
      CatPositionRec    ioCatPosition;    /*current catalog position*/
      Ptr               ioOptBuffer;      /*pointer to optional read buffer*/
      long              ioOptBufSize;     /*length of optional read buffer*/
};

typedef struct CSParam CSParam;

struct ForeignPrivParam {
      ParamBlockHeader
      long     filler1;               /*reserved*/
      long     filler2;               /*reserved*/
      Ptr      ioForeignPrivBuffer;   /*privileges data buffer*/
      long     ioForeignPrivReqCount; /*size of buffer*/
      long     ioForeignPrivActCount; /*amount of buffer used*/
      long     filler3;               /*reserved*/
      long     ioForeignPrivDirID;    /*parent directory ID of foreign */
                                      /* file or directory*/
      long     ioForeignPrivInfo1;    /*privileges data*/
      long     ioForeignPrivInfo2;    /*privileges data*/
      long     ioForeignPrivInfo3;    /*privileges data*/
      long     ioForeignPrivInfo4;    /*privileges data*/
};

typedef struct ForeignPrivParam ForeignPrivParam;
typedef ForeignPrivParam *ForeignPrivParamPtr;
```

Catalog Information Parameter Blocks

```
enum {hFileInfo, dirInfo};
typedef unsigned char CInfoType;

union CInfoPBRec {                  /*catalog information parameter block*/
      HFileInfo   hFileInfo;
      DirInfo     dirInfo;
};

typedef union CInfoPBRec CInfoPBRec;
typedef CInfoPBRec *CInfoPBPtr;

struct HFileInfo {
      ParamBlockHeader
      short             ioFRefNum;     /*file reference number*/
      char              ioFVersNum;    /*version number*/
      char              filler1;       /*reserved*/
      short             ioFDirIndex;   /*file index*/
      char              ioFlAttrib;    /*file attributes*/
      char              ioACUser;      /*directory access rights*/
      FInfo             ioFlFndrInfo;  /*information used by the Finder*/
      long              ioDirID;       /*directory ID or file ID*/
      unsigned short    ioFlStBlk;     /*first alloc. blk. of data fork*/
      long              ioFlLgLen;     /*logical EOF of data fork*/
      long              ioFlPyLen;     /*physical EOF of data fork*/
      unsigned short    ioFlRStBlk;    /*first alloc. blk. of resource fork*/
      long              ioFlRLgLen;    /*logical EOF of resource fork*/
      long              ioFlRPyLen;    /*physical EOF of resource fork*/
      unsigned long     ioFlCrDat;     /*date and time of creation*/
      unsigned long     ioFlMdDat;     /*date and time of last modification*/
      unsigned long     ioFlBkDat;     /*date and time of last backup*/
      FXInfo            ioFlXFndrInfo; /*additional Finder information*/
      long              ioFlParID;     /*file parent directory ID (integer)*/
      long              ioFlClpSiz;    /*file's clump size*/
};

typedef struct HFileInfo HFileInfo;

struct DirInfo {
      ParamBlockHeader
      short             ioFRefNum;     /*file reference number*/
      short             filler1;       /*reserved*/
      short             ioFDirIndex;   /*directory index*/
```

```
      char              ioFlAttrib;    /*directory attributes*/
      char              filler2;       /*reserved*/
      DInfo             ioDrUsrWds;    /*information used by the Finder*/
      long              ioDrDirID;     /*directory ID*/
      unsigned short    ioDrNmFls;     /*number of files in directory*/
      short             filler3[9];    /*reserved*/
      unsigned long     ioDrCrDat;     /*date and time of creation*/
      unsigned long     ioDrMdDat;     /*date and time of last modification*/
      unsigned long     ioDrBkDat;     /*date and time of last backup*/
      DXInfo            ioDrFndrInfo;  /*additional Finder information*/
      long              ioDrParID;     /*directory's parent directory ID*/
};

typedef struct DirInfo DirInfo;
```

Catalog Position Record

```
struct CatPositionRec {               /*catalog position record*/
      long     initialize;            /*starting point*/
      short    priv[6];               /*private data*/
};

typedef struct CatPositionRec CatPositionRec;
```

Catalog Move Parameter Block

```
struct CMovePBRec {                   /*catalog move parameter block*/
      QElemPtr          qLink;        /*next queue entry*/
      short             qType;        /*queue type*/
      short             ioTrap;       /*routine trap*/
      Ptr               ioCmdAddr;    /*routine address*/
      ProcPtr           ioCompletion; /*completion routine*/
      OSErr             ioResult;     /*result code*/
      StringPtr         ioNamePtr;    /*pointer to pathname*/
      short             ioVRefNum;    /*volume specification*/
      long              filler1;      /*reserved*/
      StringPtr         ioNewName;    /*name of new directory*/
      long              filler2;      /*reserved*/
      long              ioNewDirID;   /*directory ID of new directory*/
      long              filler3[2];   /*reserved*/
      long              ioDirID;      /*directory ID of current directory*/
};

typedef struct CMovePBRec CMovePBRec;
typedef CMovePBRec *CMovePBPtr;
```

Working Directory Parameter Block

```
struct WDPBRec {                        /*working directory parameter block*/
      QElemPtr          qLink;          /*next queue entry*/
      short             qType;          /*queue type*/
      short             ioTrap;         /*routine trap*/
      Ptr               ioCmdAddr;      /*routine address*/
      ProcPtr           ioCompletion;   /*completion routine*/
      OSErr             ioResult;       /*result code*/
      StringPtr         ioNamePtr;      /*pointer to pathname*/
      short             ioVRefNum;      /*volume specification*/
      short             filler1;        /*reserved*/
      short             ioWDIndex;      /*working directory index*/
      long              ioWDProcID;     /*working directory user identifier*/
      short             ioWDVRefNum;    /*working directory's vol. ref. num.*/
      short             filler2[7];     /*reserved*/
      long              ioWDDirID;      /*working directory's directory ID*/
};

typedef struct WDPBRec WDPBRec;
typedef WDPBRec *WDPBPtr;
```

File Control Block Parameter Block

```
struct FCBPBRec {                       /*file control block parameter block*/
      QElemPtr          qLink;          /*next queue entry*/
      short             qType;          /*queue type*/
      short             ioTrap;         /*routine trap*/
      Ptr               ioCmdAddr;      /*routine address*/
      ProcPtr           ioCompletion;   /*completion routine*/
      OSErr             ioResult;       /*result code*/
      StringPtr         ioNamePtr;      /*pointer to pathname*/
      short             ioVRefNum;      /*volume specification*/
      short             ioRefNum;       /*file reference number*/
      short             filler;         /*reserved*/
      short             ioFCBIndx;      /*FCB index*/
      short             filler1;        /*reserved*/
      long              ioFCBFlNm;      /*file ID*/
      short             ioFCBFlags;     /*flags*/
      unsigned short    ioFCBStBlk;     /*first allocation block of file*/
      long              ioFCBEOF;       /*logical end-of-file*/
      long              ioFCBPLen;      /*physical end-of-file*/
      long              ioFCBCrPs;      /*position of the file mark*/
      short             ioFCBVRefNum;   /*volume reference number*/
```

```
      long              ioFCBClpSiz;   /*file's clump size*/
      long              ioFCBParID;    /*parent directory ID*/
};
typedef struct FCBPBRec FCBPBRec;
typedef FCBPBRec *FCBPBPtr;
```

Volume Attributes Buffer

```
struct GetVolParmsInfoBuffer {
      short             vMVersion;        /*version number*/
      long              vMAttrib;         /*volume attributes*/
      Handle            vMLocalHand;      /*reserved*/
      long              vMServerAdr;      /*network server address*/
      long              vMVolumeGrade;    /*relative speed rating*/
      short             vMForeignPrivID;  /*foreign privilege model*/
};

typedef struct GetVolParmsInfoBuffer GetVolParmsInfoBuffer;
```

Volume Mounting Information Records

```
struct VolMountInfoHeader{              /*volume mounting information*/
      short             length;         /*length of mounting information*/
      VolumeType        media;          /*type of volume*/
};

typedef struct VolMountInfoHeader VolMountInfoHeader;
typedef VolMountInfoHeader *VolMountInfoPtr;

struct AFPVolMountInfo{                 /*AFP volume mounting information*/
      short       length;               /*length of mounting information*/
      VolumeType  media;                /*type of volume*/
      short       flags;                /*reserved; must be set to 0*/
      char        nbpInterval;          /*NBP retry interval*/
      char        nbpCount;             /*NBP retry count*/
      short       uamType;              /*user authentication method*/
      short       zoneNameOffset;       /*offset to zone name*/
      short       serverNameOffset;     /*offset server name*/
      short       volNameOffset;        /*offset to volume name*/
      short       userNameOffset;       /*offset to user name*/
      short       userPasswordOffset;   /*offset to user password*/
      short       volPasswordOffset;    /*offset to volume password*/
      char        AFPData[144];         /*standard AFP mounting info*/
};

typedef struct AFPVolMountInfo AFPVolMountInfo;
typedef AFPVolMountInfo *AFPVolMountInfoPtr;
```

Volume and File Control Blocks

```
struct VCB  {                          /*volume control block*/
      QElemPtr          qLink;         /*next queue entry*/
      short             qType;         /*queue type*/
      short             vcbFlags;      /*volume flags (bit 15 = 1 if dirty)*/
      unsigned short    vcbSigWord;    /*volume signature*/
      unsigned long     vcbCrDate;     /*date and time of volume creation*/
      unsigned long     vcbLsMod;      /*date and time of last modification*/
      short             vcbAtrb;       /*volume attributes*/
      unsigned short    vcbNmFls;      /*number of files in root directory*/
      short             vcbVBMSt;      /*first block of volume bitmap*/
      short             vcbAllocPtr;   /*start of next allocation search*/
      unsigned short    vcbNmAlBlks;   /*number of allocation blocks in */
                                       /* volume*/
      long              vcbAlBlkSiz;   /*size (in bytes) of allocation */
                                       /* blocks*/
      long              vcbClpSiz;     /*default clump size*/
      short             vcbAlBlSt;     /*first allocation block in volume*/
      long              vcbNxtCNID;    /*next unused catalog node ID*/
      unsigned short    vcbFreeBks;    /*number of unused allocation blocks*/
      Str27             vcbVN;         /*volume name*/
      short             vcbDrvNum;     /*drive number*/
      short             vcbDRefNum;    /*driver reference number*/
      short             vcbFSID;       /*file-system identifier*/
      short             vcbVRefNum;    /*volume reference number*/
      Ptr               vcbMAdr;       /*used internally*/
      Ptr               vcbBufAdr;     /*used internally*/
      short             vcbMLen;       /*used internally*/
      short             vcbDirIndex;   /*used internally*/
      short             vcbDirBlk;     /*used internally*/
      unsigned long     vcbVolBkUp;    /*date and time of last backup*/
      unsigned short    vcbVSeqNum;    /*volume backup sequence number*/
      long              vcbWrCnt;      /*volume write count*/
      long              vcbXTClpSiz;   /*clump size for extents overflow */
                                       /* file*/
      long              vcbCTClpSiz;   /*clump size for catalog file*/
      unsigned short    vcbNmRtDirs;   /*number of directories in root dir.*/
      long              vcbFilCnt;     /*number of files in volume*/
      long              vcbDirCnt;     /*number of directories in volume*/
      long              vcbFndrInfo[8];/*information used by the Finder*/
```

```
      unsigned short    vcbVCSize;     /*used internally*/
      unsigned short    vcbVBMCSiz;    /*used internally*/
      unsigned short    vcbCtlCSiz;    /*used internally*/
      unsigned short    vcbXTAlBlks;   /*size of extents overflow file*/
      unsigned short    vcbCTAlBlks;   /*size of catalog file*/
      short             vcbXTRef;      /*ref. num. for extents overflow */
                                       /* file*/
      short             vcbCTRef;      /*ref. num. for catalog file*/
      Ptr               vcbCtlBuf;     /*ptr. to extents and catalog caches*/
      long              vcbDirIDM;     /*directory last searched*/
      short             vcbOffsM;      /*offspring index at last search*/
};

typedef struct VCB VCB;

struct FCB {                          /*file control block*/
      long           fcbFlNum;        /*file ID*/
      short          fcbFlags;        /*file flags*/
      short          fcbSBlk;         /*first allocation block of file*/
      long           fcbEOF;          /*logical end-of-file*/
      long           fcbPLen;         /*physical end-of-file*/
      long           fcbCrPs;         /*current file mark position*/
      Ptr            fcbVPtr;         /*pointer to volume control block*/
      Ptr            fcbBfAdr;        /*pointer to access path buffer*/
      short          fcbFlPos;        /*unused*/
      long           fcbClmpSize;     /*file clump size*/
      Ptr            fcbBTCBPtr;      /*pointer to B*-tree control block*/
      ExtDataRec     fcbExtRec;       /*first three file extents*/
      long           fcbFType;        /*file's four Finder type bytes*/
      long           fcbCatPos;       /*catalog hint for use on Close*/
      long           fcbDirID;        /*file's parent directory ID*/
      Str31          fcbCName;        /*name of file*/
};

typedef struct FCB FCB;
```

Drive Queue Elements

```
struct DrvQEl  {                      /*drive queue element*/
      QElemPtr       qLink;           /*next queue entry*/
      short          qType;           /*flag for dQDrvSz and dQDrvSz2*/
      short          dQDrive;         /*drive number*/
      short          dQRefNum;        /*driver reference number*/
      short          dQFSID;          /*file-system identifier*/
```

```
      unsigned short dQDrvSz;       /*number of logical blocks on drive*/
      unsigned short dQDrvSz2;      /*additional field for large drives*/
};

typedef struct DrvQEl DrvQEl;
```

High-Level File Access Routines

Reading, Writing, and Closing Files

```
pascal OSErr FSRead         (short refNum, long *count, Ptr buffPtr);
pascal OSErr FSWrite        (short refNum, long *count, Ptr buffPtr);
pascal OSErr FSClose        (short refNum);
```

Manipulating the File Mark

```
pascal OSErr GetFPos        (short refNum, long *filePos);
pascal OSErr SetFPos        (short refNum, short posMode, long posOff);
```

Manipulating the End-of-File

```
pascal OSErr GetEOF         (short refNum, long *logEOF);
pascal OSErr SetEOF         (short refNum, long logEOF);
```

Allocating File Blocks

```
pascal OSErr Allocate       (short refNum, long *count);
pascal OSErr AllocContig    (short refNum, long *count);
```

Low-Level File Access Routines

Reading, Writing, and Closing Files

```
pascal OSErr PBRead         (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBReadSync     (ParmBlkPtr paramBlock);
pascal OSErr PBReadAsync    (ParmBlkPtr paramBlock);
pascal OSErr PBWrite        (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBWriteSync    (ParmBlkPtr paramBlock);
pascal OSErr PBWriteAsync   (ParmBlkPtr paramBlock);
pascal OSErr PBClose        (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBCloseSync    (ParmBlkPtr paramBlock);
pascal OSErr PBCloseAsync   (ParmBlkPtr paramBlock);
```

Manipulating the File Mark

```
pascal OSErr PBGetFPos      (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBGetFPosSync  (ParmBlkPtr paramBlock);
pascal OSErr PBGetFPosAsync (ParmBlkPtr paramBlock);
pascal OSErr PBSetFPos      (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBSetFPosSync  (ParmBlkPtr paramBlock);
pascal OSErr PBSetFPosAsync (ParmBlkPtr paramBlock);
```

Manipulating the End-of-File

```
pascal OSErr PBGetEOF       (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBGetEOFSync   (ParmBlkPtr paramBlock);
pascal OSErr PBGetEOFAsync  (ParmBlkPtr paramBlock);
pascal OSErr PBSetEOF       (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBSetEOFSync   (ParmBlkPtr paramBlock);
pascal OSErr PBSetEOFAsync  (ParmBlkPtr paramBlock);
```

Allocating File Blocks

```
pascal OSErr PBAllocate     (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBAllocateSync (ParmBlkPtr paramBlock);
pascal OSErr PBAllocateAsync(ParmBlkPtr paramBlock);
pascal OSErr PBAllocContig  (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBAllocContigSync
                            (ParmBlkPtr paramBlock);
pascal OSErr PBAllocContigAsync
                            (ParmBlkPtr paramBlock);
```

Updating Files

```
pascal OSErr PBFlushFile    (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBFlushFileSync(ParmBlkPtr paramBlock);
pascal OSErr PBFlushFileAsync
                            (ParmBlkPtr paramBlock);
```

High-Level Volume Access Routines

Unmounting Volumes

```
pascal OSErr UnmountVol     (StringPtr volName, short vRefNum);
pascal OSErr Eject          (StringPtr volName, short vRefNum);
```

Updating Volumes

```
pascal OSErr FlushVol        (StringPtr volName, short vRefNum);
```

Manipulating the Default Volume

```
pascal OSErr GetVol         (StringPtr volName, short *vRefNum);
pascal OSErr SetVol         (StringPtr volName, short vRefNum);
pascal OSErr HGetVol        (StringPtr volName, short *vRefNum,
                             long *dirID);
pascal OSErr HSetVol        (StringPtr volName, short vRefNum, long dirID);
```

Obtaining Volume Information

```
pascal OSErr GetVInfo       (short drvNum, StringPtr volName,
                             short *vRefNum, long *freeBytes);
pascal OSErr GetVRefNum     (short refNum, short *vRefNum);
```

Low-Level Volume Access Routines

Mounting and Unmounting Volumes

```
pascal OSErr PBMountVol     (ParmBlkPtr paramBlock);
pascal OSErr PBUnmountVol   (ParmBlkPtr paramBlock);
pascal OSErr PBEject        (ParmBlkPtr paramBlock);
pascal OSErr PBOffLine      (ParmBlkPtr paramBlock);
```

Updating Volumes

```
pascal OSErr PBFlushVol     (ParmBlkPtr paramBlock; Boolean async);
pascal OSErr PBFlushVolSync (ParmBlkPtr paramBlock);
pascal OSErr PBFlushVolAsync(ParmBlkPtr paramBlock);
```

Obtaining Volume Information

```
pascal OSErr PBHGetVInfo    (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHGetVInfoSync(HParmBlkPtr paramBlock);
pascal OSErr PBHGetVInfoAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBSetVInfo     (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBSetVInfoSync (HParmBlkPtr paramBlock);
pascal OSErr PBSetVInfoAsync(HParmBlkPtr paramBlock);
pascal OSErr PBHGetVolParms (HParmBlkPtr paramBlock, Boolean async);
```

```
pascal OSErr PBHGetVolParmsSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHGetVolParmsAsync
                            (HParmBlkPtr paramBlock);
```

Manipulating the Default Volume

```
pascal OSErr PBGetVol       (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBGetVolSync   (ParmBlkPtr paramBlock);
pascal OSErr PBGetVolAsync  (ParmBlkPtr paramBlock);
pascal OSErr PBSetVol       (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBSetVolSync   (ParmBlkPtr paramBlock);
pascal OSErr PBSetVolAsync  (ParmBlkPtr paramBlock);
pascal OSErr PBHGetVol      (WDPBPtr paramBlock, Boolean async);
pascal OSErr PBHGetVolSync  (WDPBPtr paramBlock);
pascal OSErr PBHGetVolAsync (WDPBPtr paramBlock);
pascal OSErr PBHSetVol      (WDPBPtr paramBlock, Boolean async);
pascal OSErr PBHSetVolSync  (WDPBPtr paramBlock);
pascal OSErr PBHSetVolAsync (WDPBPtr paramBlock);
```

File System Specification Routines

Opening Files

```
pascal OSErr FSpOpenDF      (const FSSpec *spec, char permission,
                             short *refNum);
pascal OSErr FSpOpenRF      (const FSSpec *spec, char permission,
                             short *refNum);
```

Creating and Deleting Files and Directories

```
pascal OSErr FSpCreate      (const FSSpec *spec, OSType creator,
                             OSType fileType, ScriptCode scriptTag);
pascal OSErr FSpDirCreate   (const FSSpec *spec, ScriptCode scriptTag,
                             long *createdDirID);
pascal OSErr FSpDelete      (const FSSpec *spec);
```

Accessing Information About Files and Directories

```
pascal OSErr FSpGetFInfo    (const FSSpec *spec, FInfo *fndrInfo);
pascal OSErr FSpSetFInfo    (const FSSpec *spec, const FInfo *fndrInfo);
pascal OSErr FSpSetFLock    (const FSSpec *spec);
pascal OSErr FSpRstFLock    (const FSSpec *spec);
pascal OSErr FSpRename      (const FSSpec *spec, ConstStr255Param newName);
```

Moving Files or Directories

```
pascal OSErr FSpCatMove     (const FSSpec *source, const FSSpec *dest);
```

Exchanging the Data in Two Files

```
pascal OSErr FSpExchangeFiles
                            (const FSSpec *source, const FSSpec *dest);
```

Creating File System Specifications

```
pascal OSErr FSMakeFSSpec   (short vRefNum, long dirID,
                             ConstStr255Param fileName, FSSpecPtr spec);
pascal OSErr PBMakeFSSpec   (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBMakeFSSpecSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBMakeFSSpecAsync
                            (HParmBlkPtr paramBlock);
```

High-Level HFS Routines

Opening Files

```
pascal OSErr HOpenDF         (short vRefNum, long dirID,
                              const Str255 fileName, char permission,
                              short *refNum);
pascal OSErr HOpenRF         (short vRefNum, long dirID,
                              const Str255 fileName, char permission,
                              short *refNum);
pascal OSErr HOpen           (short vRefNum, long dirID,
                              const Str255 fileName, char permission,
                              short *refNum);
```

Creating and Deleting Files and Directories

```
pascal OSErr HCreate         (short vRefNum, long dirID,
                              const Str255 fileName, OSType creator,
                              OSType fileType);
pascal OSErr DirCreate       (short vRefNum, long parentDirID,
                              const Str255 directoryName,
                              long *createdDirID);
pascal OSErr HDelete         (short vRefNum, long dirID,
                              const Str255 fileName);
```

Accessing Information About Files and Directories

```
pascal OSErr HGetFInfo       (short vRefNum, long dirID,
                              const Str255 fileName, FInfo *fndrInfo);
pascal OSErr HSetFInfo       (short vRefNum, long dirID,
                              const Str255 fileName, const FInfo *fndrInfo);
pascal OSErr HSetFLock       (short vRefNum, long dirID,
                              const Str255 fileName);
pascal OSErr HRstFLock       (short vRefNum, long dirID,
                              const Str255 fileName);
pascal OSErr HRename         (short vRefNum, long dirID,
                              const Str255 oldName, const Str255 newName);
```

Moving Files or Directories

```
pascal OSErr CatMove         (short vRefNum, long dirID,
                              const Str255 oldName, long newDirID,
                              const Str255 newName);
```

Maintaining Working Directories

```
pascal OSErr OpenWD         (short vRefNum, long dirID, long procID,
                             short *wdRefNum);
pascal OSErr CloseWD        (short wdRefNum);
pascal OSErr GetWDInfo      (short wdRefNum, short *vRefNum, long *dirID,
                             long *procID);
```

Low-Level HFS Routines

Opening Files

```
pascal OSErr PBHOpenDF      (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHOpenDFSync  (HParmBlkPtr paramBlock);
pascal OSErr PBHOpenDFAsync (HParmBlkPtr paramBlock);
pascal OSErr PBHOpenRF      (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHOpenRFSync  (HParmBlkPtr paramBlock);
pascal OSErr PBHOpenRFAsync (HParmBlkPtr paramBlock);
pascal OSErr PBHOpen        (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHOpenSync    (HParmBlkPtr paramBlock);
pascal OSErr PBHOpenAsync   (HParmBlkPtr paramBlock);
```

Creating and Deleting Files and Directories

```
pascal OSErr PBHCreate      (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHCreateSync  (HParmBlkPtr paramBlock);
pascal OSErr PBHCreateAsync (HParmBlkPtr paramBlock);
pascal OSErr PBDirCreate    (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBDirCreateSync(HParmBlkPtr paramBlock);
pascal OSErr PBDirCreateAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHDelete      (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHDeleteSync  (HParmBlkPtr paramBlock);
pascal OSErr PBHDeleteAsync (HParmBlkPtr paramBlock);
```

Accessing Information About Files and Directories

```
pascal OSErr PBGetCatInfo   (CInfoPBPtr paramBlock, Boolean async);
pascal OSErr PBGetCatInfoSync
                            (CInfoPBPtr paramBlock, Boolean async);
pascal OSErr PBGetCatInfoAsync
                            (CInfoPBPtr paramBlock);
pascal OSErr PBSetCatInfo   (CInfoPBPtr paramBlock, Boolean async);
pascal OSErr PBSetCatInfoSync
                            (CInfoPBPtr paramBlock);
pascal OSErr PBSetCatInfoAsync
                            (CInfoPBPtr paramBlock);
pascal OSErr PBHGetFInfo    (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHGetFInfoSync(HParmBlkPtr paramBlock);
pascal OSErr PBHGetFInfoAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHSetFInfo    (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHSetFInfoSync(HParmBlkPtr paramBlock);
pascal OSErr PBHSetFInfoAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHSetFLock    (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHSetFLockSync(HParmBlkPtr paramBlock);
pascal OSErr PBHSetFLockAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHRstFLock    (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHRstFLockSync(HParmBlkPtr paramBlock);
pascal OSErr PBHRstFLockAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHRename      (HParmBlkPtr paramBlock, Boolean async);
```

```
pascal OSErr PBHRenameSync  (HParmBlkPtr paramBlock);
pascal OSErr PBHRenameAsync (HParmBlkPtr paramBlock);
```

Moving Files or Directories

```
pascal OSErr PBCatMove      (CMovePBPtr paramBlock, Boolean async);
pascal OSErr PBCatMoveSync  (CMovePBPtr paramBlock);
pascal OSErr PBCatMoveAsync (CMovePBPtr paramBlock);
```

Maintaining Working Directories

```
pascal OSErr PBOpenWD       (WDPBPtr paramBlock, Boolean async);
pascal OSErr PBOpenWDSync   (WDPBPtr paramBlock);
pascal OSErr PBOpenWDAsync  (WDPBPtr paramBlock);
pascal OSErr PBCloseWD      (WDPBPtr paramBlock, Boolean async);
pascal OSErr PBCloseWDSync  (WDPBPtr paramBlock);
pascal OSErr PBCloseWDAsync (WDPBPtr paramBlock);
pascal OSErr PBGetWDInfo    (WDPBPtr paramBlock, Boolean async);
pascal OSErr PBGetWDInfoSync(WDPBPtr paramBlock);
pascal OSErr PBGetWDInfoAsync
                            (WDPBPtr paramBlock);
```

Searching a Catalog

```
pascal OSErr PBCatSearch    (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBCatSearchSync(HParmBlkPtr paramBlock);
pascal OSErr PBCatSearchAsync
                            (HParmBlkPtr paramBlock);
```

Exchanging the Data in Two Files

```
pascal OSErr PBExchangeFiles(HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBExchangeFilesSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBExchangeFilesAsync
                            (HParmBlkPtr paramBlock);
```

Shared Environment Routines

Opening Files While Denying Access

```
pascal OSErr PBHOpenDeny    (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHOpenDenySync(HParmBlkPtr paramBlock);
```

```
pascal OSErr PBHOpenDenyAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHOpenRFDeny  (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHOpenRFDenySync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHOpenRFDenyAsync
                            (HParmBlkPtr paramBlock);
```

Locking and Unlocking File Ranges

```
pascal OSErr PBLockRange    (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBLockRangeSync(ParmBlkPtr paramBlock);
pascal OSErr PBLockRangeAsync
                            (ParmBlkPtr paramBlock);
pascal OSErr PBUnlockRange  (ParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBUnlockRangeSync
                            (ParmBlkPtr paramBlock);
pascal OSErr PBUnlockRangeAsync
                            (ParmBlkPtr paramBlock);
```

Manipulating Share Points

```
pascal OSErr PBShare        (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBShareSync    (HParmBlkPtr paramBlock);
pascal OSErr PBShareAsync   (HParmBlkPtr paramBlock);
pascal OSErr PBUnshare      (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBUnshareSync  (HParmBlkPtr paramBlock);
pascal OSErr PBUnshareAsync (HParmBlkPtr paramBlock);
pascal OSErr PBGetUGEntry   (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBGetUGEntrySync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBGetUGEntryAsync
                            (HParmBlkPtr paramBlock);
```

Controlling Directory Access

```
pascal OSErr PBHGetDirAccess(HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHGetDirAccessSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHGetDirAccessAsync
                            (HParmBlkPtr paramBlock);
```

```
pascal OSErr PBHSetDirAccess(HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHSetDirAccessSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHSetDirAccessAsync
                            (HParmBlkPtr paramBlock);
```

Mounting Volumes

```
pascal OSErr PBGetVolMountInfoSize
                            (ParmBlkPtr paramBlock);
pascal OSErr PBGetVolMountInfo
                            (ParmBlkPtr paramBlock);
pascal OSErr PBVolumeMount  (ParmBlkPtr paramBlock);
```

Controlling Login Access

```
pascal OSErr PBHGetLogInInfo(HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHGetLogInInfoSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHGetLogInInfoAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHMapID       (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHMapIDSync   (HParmBlkPtr paramBlock);
pascal OSErr PBHMapIDAsync  (HParmBlkPtr paramBlock);
pascal OSErr PBHMapName     (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHMapNameSync (HParmBlkPtr paramBlock);
pascal OSErr PBHMapNameAsync(HParmBlkPtr paramBlock);
```

Copying and Moving Files

```
pascal OSErr PBHCopyFile    (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHCopyFileSync(HParmBlkPtr paramBlock);
pascal OSErr PBHCopyFileAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHMoveRename  (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBHMoveRenameSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBHMoveRenameAsync
                            (HParmBlkPtr paramBlock);
```

File ID Routines

Resolving File ID References

```
pascal OSErr PBResolveFileIDRef
                            (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBResolveFileIDRefSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBResolveFileIDRefAsync
                            (HParmBlkPtr paramBlock);
```

Creating and Deleting File ID References

```
pascal OSErr PBCreateFileIDRef
                            (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBCreateFileIDRefSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBCreateFileIDRefAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBDeleteFileIDRef
                            (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBDeleteFileIDRefSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBDeleteFileIDRefAsync
                            (HParmBlkPtr paramBlock);
```

Foreign File System Routines

Accessing Privilege Information in Foreign File Systems

```
pascal OSErr PBGetForeignPrivs
                            (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBGetForeignPrivsSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBGetForeignPrivsAsync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBSetForeignPrivs
                            (HParmBlkPtr paramBlock, Boolean async);
pascal OSErr PBSetForeignPrivsSync
                            (HParmBlkPtr paramBlock);
pascal OSErr PBSetForeignPrivsAsync
                            (HParmBlkPtr paramBlock);
```

Utility Routines

Obtaining Queue Headers

```
#define GetFSQHdr()         (QHdrPtr);
#define GetVCBQHdr()        (QHdrPtr);
#define GetDrvQHdr()        (QHdrPtr);
```

Adding a Drive

```
pascal void AddDrive        (short drvrRefNum, short drvNum, DrvQElPtr qEl);
```

Obtaining File Control Block Information

```
pascal OSErr PBGetFCBInfo   (FCBPBPtr paramBlock, Boolean async);
pascal OSErr PBGetFCBInfoSync
                            (FCBPBPtr paramBlock);
pascal OSErr PBGetFCBInfoAsync
                            (FCBPBPtr paramBlock);
```

Application-Defined Routine

Completion Routines

```
pascal void MyCompletionProc(void);
```

Assembly-Language Summary

Constants

```
;flags in trap words
hfsBit            EQU      9        ;set for an HFS call
asyncTrpBit       EQU      10       ;set for an asynchronous call

;masks for flags in trap words
newHFS            EQU      $200     ;make an HFS call
ASYNC             EQU      $400     ;make an asynchronous call
```

Data Structures

File System Specification Record

0	vRefNum	word	volume reference number
2	parID	long	parent directory ID
6	name	64 bytes	filename or directory name

HFS Parameter Block Common Fields

0	qLink	long	next queue entry
4	qType	word	queue type
6	ioTrap	word	routine trap
8	ioCmdAddr	long	routine address
12	ioCompletion	long	address of completion routine
16	ioResult	word	result code
18	ioNamePtr	long	pointer to pathname
22	ioVRefNum	word	volume specification

I/O Parameter Variant

24	ioRefNum	word	file reference number
26	ioVersNum	byte	version number
27	ioPermssn	byte	read / write permission
28	ioMisc	long	miscellaneous
32	ioBuffer	long	data buffer
36	ioReqCount	long	requested number of bytes
40	ioActCount	long	actual number of bytes
44	ioPosMode	word	positioning mode and newline character
46	ioPosOffset	long	positioning offset

File Parameter Variant

24	ioFRefNum	word	file reference number
26	ioFVersNum	byte	file version number (unused)
27	filler1	byte	reserved
28	ioFDirIndex	word	directory index
30	ioFlAttrib	byte	file attributes
31	ioFlVersNum	byte	file version number (unused)
32	ioFlFndrInfo	16 bytes	information used by the Finder
48	ioDirID	long	directory ID or file ID
52	ioFlStBlk	word	first allocation block of data fork
54	ioFlLgLen	long	logical end-of-file of data fork
58	ioFlPyLen	long	physical end-of-file of data fork
62	ioFlRStBlk	word	first allocation block of resource fork
64	ioFlRLgLen	long	logical end-of-file of resource fork
68	ioFlRPyLen	long	physical end-of-file of resource fork
72	ioFlCrDat	long	date and time of creation
76	ioFlMdDat	long	date and time of last modification

Volume Parameter Variant

24	filler2	long	reserved
28	ioVolIndex	word	volume index
30	ioVCrDate	long	date and time of initialization
34	ioVLsMod	long	date and time of last modification
38	ioVAtrb	word	volume attributes
40	ioVNmFls	word	number of files in root directory
42	ioVBitMap	word	first block of volume bitmap
44	ioAllocPtr	word	first block of next new file
46	ioVNmAlBlks	word	number of allocation blocks
48	ioVAlBlkSiz	long	size of allocation blocks
50	ioVClpSiz	long	default clump size
54	ioAlBlSt	word	first block in volume map
56	ioVNxtCNID	long	next unused node ID
60	ioVFrBlk	word	number of unused allocation blocks
62	ioVSigWord	word	volume signature
64	ioVDrvInfo	word	drive number
66	ioVDRefNum	word	driver reference number
68	ioVFSID	word	file-system identifier
70	ioVBkUp	long	date and time of last backup
74	ioVSeqNum	word	used internally
76	ioVWrCnt	long	volume write count
80	ioVFilCnt	long	number of files on volume
84	ioVDirCnt	long	number of directories on volume
88	ioVFndrInfo	32 bytes	information used by the Finder

Access Variant

24	filler3	word	reserved
26	ioDenyModes	word	access mode information
28	filler4	word	reserved
30	filler5	byte	reserved
31	ioACUser	byte	user access rights
32	filler6	long	reserved
36	ioACOwnerID	long	owner ID
40	ioACGroupID	long	group ID
44	ioACAccess	long	directory access rights

Object Variant

24	filler7	word	reserved
26	ioObjType	word	function code
28	ioObjNamePtr	long	pointer to returned creator / group name
32	ioObjID	long	creator / group ID

Copy Variant

24	ioDstVRefNum	word	destination volume identifier
26	filler8	word	reserved
28	ioNewName	long	pointer to destination pathname
32	ioCopyName	long	pointer to optional name
36	ioNewDirID	long	directory ID of destination directory

Working Directory Variant

24	filler9	word	reserved
26	ioWDIndex	word	working directory's index
28	ioWDProcID	long	working directory's user identifier
32	ioWDVRefNum	word	working directory's volume reference number
34	filler10	word	reserved
36	filler11	long	reserved
40	filler12	long	reserved
44	filler13	long	reserved
48	ioWDDirID	long	working directory's directory ID

File ID Variant

24	filler14	long	reserved
28	ioDestNamePtr	long	pointer to destination filename
32	filler15	long	reserved
36	ioDestDirID	long	destination parent directory ID
40	filler16	long	reserved
44	filler17	long	reserved
48	ioSrcDirID	long	source parent directory ID
52	filler18	word	reserved
54	ioFileID	long	file ID

Catalog Search Variant

24	ioMatchPtr	long	pointer to array of matches
28	ioReqMatchCount	long	maximum match count
32	ioActMatchCount	long	actual match count
36	ioSearchBits	long	search criteria selector
40	ioSearchInfo1	long	pointer to values and lower bounds
44	ioSearchInfo2	long	pointer to masks and upper bounds
48	ioSearchTime	long	time limit on search
52	ioCatPosition	16 bytes	catalog position record
68	ioOptBuffer	long	pointer to optional read buffer
72	ioOptBufSize	long	length of optional read buffer

Foreign Privileges Variant

24	filler21	long	reserved
28	filler22	long	reserved
32	ioForeignPrivBuffer	long	pointer to privileges data buffer
36	ioForeignPrivReqCount	long	size allocated for buffer
40	ioForeignPrivActCount	long	amount of buffer used
44	filler23	long	reserved
48	ioForeignPrivDirID	long	parent directory ID of target
52	ioForeignPrivInfo1	long	privileges data
56	ioForeignPrivInfo2	long	privileges data
60	ioForeignPrivInfo3	long	privileges data
64	ioForeignPrivInfo4	long	privileges data

Catalog Information Parameter Block (Files Variant)

24	ioFRefNum	word	file reference number
26	ioFVersNum	byte	version number
27	filler1	byte	reserved
28	ioFDirIndex	word	directory index
30	ioFlAttrib	byte	file attributes
31	ioACUser	byte	directory access rights
32	ioFlUsrWds	16 bytes	information used by the Finder
48	ioFlNum	long	file ID
52	ioFlStBlk	word	first allocation block of data fork
54	ioFlLgLen	long	logical end-of-file of data fork
58	ioFlPyLen	long	physical end-of-file of data fork
62	ioFlRStBlk	word	first allocation block of resource fork
64	ioFlRLgLen	long	logical end-of-file of resource fork
68	ioFlRPyLen	long	physical end-of-file of resource fork
72	ioFlCrDat	long	date and time of creation
76	ioFlMdDat	long	date and time of last modification
80	ioFlBkDat	long	date and time of last backup
84	ioFlXFndrInfo	16 bytes	additional information used by the Finder
100	ioFlParID	long	file parent directory ID
104	ioFlClpSiz	long	file's clump size

Catalog Information Parameter Block (Directories Variant)

24	ioFRefNum	word	file reference number
26	ioFVersNum	byte	version number
27	filler1	byte	reserved
28	ioFDirIndex	word	directory index
30	ioFlAttrib	byte	directory attributes
31	ioACUser	byte	directory access rights
32	ioDrUsrWds	16 bytes	information used by the Finder
48	ioDrDirID	long	directory ID
52	ioDrNmFls	word	number of files in directory
54	filler3	18 bytes	reserved
72	ioDrCrDat	long	date and time of creation
76	ioDrMdDat	long	date and time of last modification
80	ioDrBkDat	long	date and time of last backup
84	ioDrFndrInfo	16 bytes	additional information used by the Finder
100	ioDrParID	long	directory's parent directory ID

Catalog Position Record

0	initialize	long	starting place for next search
4	priv	12 bytes	private data

Catalog Move Parameter Block

24	filler1	long	reserved
28	ioNewName	long	pointer to name of new directory
32	filler2	long	reserved
36	ioNewDirID	long	directory ID of new directory
40	filler3	8 bytes	reserved
48	ioDirID	long	directory ID of current directory

Working Directory Parameter Block

24	filler1	word	reserved
26	ioWDIndex	word	working directory's index
28	ioWDProcID	long	working directory's user identifier
32	ioWDVRefNum	word	working directory's volume reference number
34	filler2	14 bytes	reserved
48	ioWDDirID	long	working directory's directory ID

File Control Block Parameter Block

24	ioRefNum	word	file reference number
26	filler	word	reserved
28	ioFCBIndx	word	FCB index
30	ioFCBfiller1	word	reserved
32	ioFCBFlNm	long	file ID
36	ioFCBFlags	word	flags
38	ioFCBStBlk	word	first allocation block of file
40	ioFCBEOF	long	logical end-of-file
44	ioFCBPLen	long	physical end-of-file
48	ioFCBCrPs	long	position of the file mark
52	ioFCBVRefNum	word	volume reference number
54	ioFCBClpSiz	long	file's clump size
58	ioFCBParID	long	parent directory ID

Volume Attributes Buffer

0	vMVersion	word	version number
2	vMAttrib	long	volume attributes
6	vMLocalHand	long	reserved
10	vMServerAdr	long	network server address
14	vMVolumeGrade	long	relative speed rating
18	vMForeignPrivID	word	foreign privilege model

Volume Mounting Information Record

0	length	word	length of record
2	media	4 bytes	type of volume

AFP Mounting Information Record

0	length	word	length of record
2	media	4 bytes	type of volume
6	flags	word	reserved; must be 0
8	nbpInterval	byte	NBP retry interval
9	nbpCount	byte	NBP retry count
10	uamType	word	user authentication method
12	zoneNameOffset	word	offset to zone name
14	serverNameOffset	word	offset to server name
16	volNameOffset	word	offset to volume name
18	userNameOffset	word	offset to user name
20	userPasswordOffset	word	offset to user password
22	volPasswordOffset	word	offset to volume password
24	AFPData	144 bytes	mounting data

Volume Control Block Data Structure (Internal)

0	qLink	long	next queue entry
4	qType	word	queue type
6	vcbFlags	word	volume flags
8	vcbSigWord	word	volume signature
10	vcbCrDate	long	date and time of initialization
14	vcbLsMod	long	date and time of last modification
18	vcbAtrb	word	volume attributes
20	vcbNmFls	word	number of files in root directory
22	vcbVBMSt	word	first block of volume bitmap
24	vcbAllocPtr	word	start of next allocation search
26	vcbNmAlBlks	word	number of allocation blocks in volume
28	vcbAlBlkSiz	long	size (in bytes) of allocation block
32	vcbClpSiz	long	default clump size
36	vcbAlBlSt	word	first allocation block in volume
38	vcbNxtCNID	long	next unused catalog node ID
42	vcbFreeBks	word	number of unused allocation blocks
44	vcbVN	28 bytes	volume name preceded by length byte
72	vcbDrvNum	word	drive number
74	vcbDRefNum	word	driver reference number
76	vcbFSID	word	file-system identifier
78	vcbVRefNum	word	volume reference number
80	vcbMAdr	long	pointer to block map
84	vcbBufAdr	long	pointer to volume buffer
88	vcbMLen	word	number of bytes in block map
90	vcbDirIndex	word	reserved
92	vcbDirBlk	word	reserved
94	vcbVolBkUp	long	date and time of last backup
98	vcbVSeqNum	word	volume backup sequence number

100	vcbWrCnt	long	volume write count
104	vcbXTClpSiz	long	clump size for extents overflow file
108	vcbCTClpSiz	long	clump size for catalog file
112	vcbNmRtDirs	word	number of directories in root directory
114	vcbFilCnt	long	number of files in volume
118	vcbDirCnt	long	number of directories in volume
122	vcbFndrInfo	32 bytes	information used by the Finder
154	vcbVCSize	word	reserved
156	vcbVBMCSiz	word	reserved
158	vcbCtlCSiz	word	reserved
160	vcbXTAlBks	word	size in blocks of extents overflow file
162	vcbCTAlBks	word	size in blocks of catalog file
164	vcbXTRef	word	file reference number for extents overflow file
166	vcbCTRef	word	file reference number for catalog file
168	vcbCtlBuf	long	pointer to extents and catalog tree caches
172	vcbDirIDM	long	directory last searched
176	vcbOffsM	word	offspring index at last search

File Control Block Data Structure (Internal)

0	fcbFlNum	long	file ID
4	fcbFlags	word	file flags
6	fcbSBlk	word	first allocation block of file
8	fcbEOF	long	logical end-of-file
12	fcbPLen	long	physical end-of-file
16	fcbCrPs	long	current file mark position
20	fcbVPtr	long	pointer to volume control block
24	fcbBfAdr	long	pointer to access path buffer
28	fcbFlPos	word	reserved
30	fcbClmpSize	long	file's clump size
34	fcbBTCBPtr	long	pointer to B*-tree control block
38	fcbExtRec	12 bytes	first three file extents
50	fcbFType	long	file's four Finder type bytes
54	fcbCatPos	long	catalog hint for use on close
58	fcbDirID	long	file's parent directory ID
62	fcbCName	32 bytes	name of open file, preceded by length byte

Drive Queue Elements

0	qLink	long	next queue entry
4	qType	word	flag for dQDrvSz and dQDrvSz2 fields
6	dQDrive	word	drive number
8	dQRefNum	word	driver reference number
10	dQFSID	word	file-system identifier
12	dQDrvSz	word	number of logical blocks on drive
14	dQDrvSz2	word	additional field for large drives

Trap Macros

Trap Macro Names

Pascal name	Trap macro name
PBAllocate	_Allocate
PBAllocContig	_AllocContig
PBClose	_Close
PBDirCreate	_DirCreate
PBEject	_Eject
PBFlushFile	_FlushFile
PBFlushVol	_FlushVol
PBGetEOF	_GetEOF
PBGetFPos	_GetFPos
PBGetVol	_GetVol
PBHCreate	_HCreate
PBHDelete	_HDelete
PBHGetFInfo	_HGetFileInfo
PBHGetVInfo	_HGetVolInfo
PBHGetVol	_HGetVol
PBHGetVolParms	_GetVolParms
PBHOpen	_HOpen
PBHOpenRF	_HOpenRF
PBHRename	_HRename
PBHRstFLock	_HRstFLock
PBHSetFInfo	_HSetFileInfo
PBHSetFLock	_HSetFLock
PBHSetVol	_HSetVol
PBMountVol	_MountVol
PBOffLine	_OffLine
PBRead	_Read
PBSetEOF	_SetEOF
PBSetFPos	_SetFPos
PBSetVInfo	_SetVolInfo
PBSetVol	_SetVol
PBUnmountVol	_UnmountVol
PBWrite	_Write

Trap Macros Requiring Routine Selectors

_HFSDispatch

Selector	Routine
$0001	PBOpenWD
$0002	PBCloseWD
$0005	PBCatMove
$0006	PBDirCreate
$0007	PBGetWDInfo
$0008	PBGetFCBInfo
$0009	PBGetCatInfo
$000A	PBSetCatInfo
$000B	PBSetVInfo
$0010	PBLockRange
$0011	PBUnlockRange
$0014	PBCreateFileIDRef
$0015	PBDeleteFileIDRef
$0016	PBResolveFileIDRef
$0017	PBExchangeFiles
$0018	PBCatSearch
$001A	PBHOpenDF
$001B	PBMakeFSSpec
$0030	PBHGetVolParms
$0031	PBHGetLogInInfo
$0032	PBHGetDirAccess
$0033	PBHSetDirAccess
$0034	PBHMapID
$0035	PBHMapName
$0036	PBHCopyFile
$0037	PBHMoveRename
$0038	PBHOpenDeny
$0039	PBHOpenRFDeny
$003F	PBGetVolMountInfoSize
$0040	PBGetVolMountInfo
$0041	PBVolumeMount
$0042	PBShare
$0043	PBUnshare

Selector	Routine
$0044	PBGetUGEntry
$0060	PBGetForeignPrivs
$0061	PBSetForeignPrivs

_HighLevelFSDispatch

Selector	Routine
$0001	FSMakeFSSpec
$0002	FSpOpenDF
$0003	FSpOpenRF
$0004	FSpCreate
$0005	FSpDirCreate
$0006	FSpDelete
$0007	FSpGetFInfo
$0008	FSpSetFInfo
$0009	FSpSetFLock
$000A	FSpRstFLock
$000B	FSpRename
$000C	FSpCatMove
$000D	FSpOpenResFile
$000E	FSpCreateResFile
$000F	FSpExchangeFiles

Global Variables

BootDrive	word	Working directory reference number for startup volume.
DefVCBPtr	long	Pointer to default volume control block.
DrvQHdr	10 bytes	Drive queue header.
FSFCBLen	word	Size of a file control block.
FSQHdr	10 bytes	File I/O queue header.
ToExtFS	long	Pointer to external file system.
VCBQHdr	10 bytes	Volume control block queue header.

Result Codes

noErr	0	No error
notOpenErr	–28	AppleTalk is not open
dirFulErr	–33	File directory full
dskFulErr	–34	All allocation blocks on the volume are full
nsvErr	–35	Volume not found
ioErr	–36	I/O error
bdNamErr	–37	Bad filename or volume name
fnOpnErr	–38	File not open
eofErr	–39	Logical end-of-file reached
posErr	–40	Attempt to position mark before start of file
tmfoErr	–42	Too many files open
fnfErr	–43	File not found
wPrErr	–44	Hardware volume lock
fLckdErr	–45	File is locked
vLckdErr	–46	Software volume lock
fBsyErr	–47	File is busy; one or more files are open; directory not empty or working directory control block is open
dupFNErr	–48	A file with the specified name already exists
opWrErr	–49	File already open for writing
paramErr	–50	Parameter error
rfNumErr	–51	Reference number specifies nonexistent access path; bad working directory reference number
gfpErr	–52	Error during GetFPos
volOfflinErr	–53	Volume is offline
permErr	–54	Attempt to open locked file for writing
volOnLinErr	–55	Specified volume is already mounted and online
nsDrvErr	–56	Specified drive number doesn't match any number in the drive queue
noMacDskErr	–57	Volume lacks Macintosh-format directory
extFSErr	–58	External file system
fsRnErr	–59	Problem during rename
badMDBErr	–60	Bad master directory block
wrPermErr	–61	Read/write permission doesn't allow writing
memFullErr	–108	Insufficient memory available
dirNFErr	–120	Directory not found
tmwdoErr	–121	Too many working directories open
badMovErr	–122	Attempted to move into offspring
wrgVolTypErr	–123	Not an HFS volume

volGoneErr	–124	Server volume has been disconnected
fsDSIntErr	–127	Internal file system error
fidNotFoundErr	–1300	File ID not found
fidExists	–1301	File ID already exists
notAFileErr	–1302	Specified file is a directory
diffVolErr	–1303	Files are on different volumes
catChangedErr	–1304	Catalog has changed and catalog position record may be invalid
sameFileErr	–1306	Files are the same
afpAccessDenied	–5000	The operation has failed because the user does not have the correct access to the file or folder
afpBadUAM	–5002	User authentication method is unknown
afpBadVersNum	–5003	Workstation is using an AFP version that the server doesn't recognize
afpDenyConflict	–5006	The operation has failed because the permission or deny mode conflicts with the mode in which the fork has already been opened
afpNoMoreLocks	–5015	Byte range locking has failed because the server cannot lock any additional ranges
afpNoServer	–5016	Server is not responding
afpRangeNotLocked	–5020	User has attempted to unlock a range that was not locked by that user
afpRangeOverlap	–5021	User attempted to lock some or all of a range that is already locked
afpUserNotAuth	–5023	User authentication failed (usually, password is not correct)
afpObjectTypeErr	–5025	Object was a file, not a directory; or, this directory is not a share point
afpContainsSharedErr	–5033	The directory contains a share point
afpIDNotFound	–5034	File ID not found
afpIDExists	–5035	File ID already exists
afpCatalogChanged	–5037	Catalog has changed and search cannot be resumed
afpSameObjectErr	–5038	Source and destination are the same
afpBadIDErr	–5039	Bad file ID
afpPwdExpired	–5042	Password has expired on server
afpInsideSharedErr	–5043	The directory is inside a shared directory
afpBadDirIDType	–5060	Not a fixed directory ID volume
afpCantMountMoreSrvrs	–5061	Maximum number of volumes have been mounted
afpAlreadyMounted	–5062	Volume already mounted
afpSameNodeErr	–5063	Attempt to log on to a server running on the same machine

Standard File Package

Contents

This chapter describes how your application can use the Standard File Package to manage the user interface for naming and identifying files. The Standard File Package displays the dialog boxes that let the user specify the names and locations of files to be saved or opened, and it reports the user's choices to your application.

The Standard File Package supports both standard and customized dialog boxes. The standard dialog boxes are sufficient for applications that do not require additional controls or other elements in the user interface. The chapter "Introduction to File Management" earlier in this book provides a detailed description of how to display the standard dialog boxes by calling two of the enhanced Standard File Package routines introduced in system software version 7.0. You need to read this chapter if your application needs to use features not described in that earlier chapter (such as customized dialog boxes or a special file filter function). You also need to read this chapter if you want your application to run in an environment where the new routines are not available and your development system does not provide glue code that allows you to call the enhanced routines in earlier system software versions.

To use this chapter, you should be familiar with the Dialog Manager, the Control Manager, and the Finder. You need to know about the Dialog Manager if you want to provide a modal-dialog filter function that handles events received from the Event Manager before they are passed to the `ModalDialog` procedure (which the Standard File Package uses to manage both standard and customized dialog boxes). You need to know about the Control Manager if you want to customize the user interface by adding controls (such as radio buttons or pop-up menus). You need to know about the Finder if your application supports stationery documents. See the appropriate chapters in *Inside Macintosh: Macintosh Toolbox Essentials* for specific information about these system software components.

This chapter provides an introduction to the Standard File Package and then discusses

- how you can display the standard file selection dialog boxes
- how the Standard File Package interprets user actions in those dialog boxes
- how to manage customized dialog boxes
- how to set the directory whose contents are listed in a dialog box
- how to allow the user to select a volume or directory
- how to use the original Standard File Package routines

About the Standard File Package

Macintosh applications typically have a File menu from which the user can save and open documents, via the Save, Save As, and Open commands. When the user chooses Open to open an existing document, your application needs to determine which document to open. Similarly, when the user chooses Save As, or Save when the document is untitled, your application needs to ask the user for the name and location of the file in which the document is to be saved.

The Standard File Package provides a number of routines that handle the user interface between the user and your application when the user saves or opens a document. It displays dialog boxes through which the user specifies the name and location of the document to be saved or opened. It also allows your application to customize the dialog boxes and, through callback routines, to handle user actions during the dialogs. The Standard File Package procedures return information about the user's choices to your application through a reply record.

The Standard File Package is available in all versions of system software. However, significant improvements were made to the package in system software version 7.0. The Standard File Package in version 7.0 introduces

- a pair of simplified procedures (`StandardGetFile` and `StandardPutFile`) that you call to display and handle the standard Open and Save dialog boxes
- a pair of customizable procedures (`CustomGetFile` and `CustomPutFile`) that you call when you need more control over the interaction
- a new reply record (`StandardFileReply`) that identifies files and folders with a file system specification record and that accommodates the new Finder features introduced in system software version 7.0
- a new layout for the standard dialog boxes

This section describes in detail the standard and customized user interfaces provided by the enhanced Standard File Package in system software version 7.0 and later. If your application is to run in earlier system software versions as well, you should read the section "Using the Original Procedures" on page 3-40.

IMPORTANT

If you use the enhanced routines introduced in system software version 7.0, you must also support the Open Documents Apple event. ▲

Standard User Interfaces

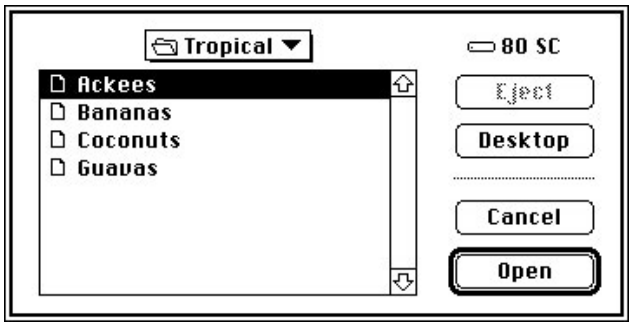
If your application has no special interface requirements, you can use the `StandardGetFile` and `StandardPutFile` procedures to display the standard dialog boxes for opening and saving documents.

Opening Files

You use the `StandardGetFile` procedure when you want to let the user select a file to be opened. Figure 3-1 illustrates a sample dialog box displayed by `StandardGetFile`.

The directory whose contents are listed in the **display list** in the dialog box displayed by `StandardGetFile` is known as the **current directory.** In Figure 3-1, the current directory is named "Tropical." The user can change the current directory in several ways. To ascend the directory hierarchy from the current directory, the user can click the directory pop-up menu and select a new directory from among those in the menu. To ascend one level of the directory hierarchy, the user can click the volume icon. To ascend immediately to the top of the directory hierarchy, the user can click the Desktop button.

Figure 3-1 The default Open dialog box



To descend the directory hierarchy, the user can double-click any of the folder names in the list (or select a folder by clicking its name once and then clicking the Open button). Whenever the current directory changes, the list of folders and files is updated to reflect the contents of the new current directory.

The volume on which the current directory is located is the **current volume** (or **current disk**), whose name is displayed to the right of the directory pop-up menu. If the current volume is a removable volume, the Eject button is active. The user can click Eject to eject the current volume and insert another, which then becomes the current volume. If the user inserts an uninitialized or otherwise unreadable disk, the Standard File Package calls the Disk Initialization Manager to provide the standard user interface for initializing and naming a disk. See the chapter "Disk Initialization Manager" in this book for details.

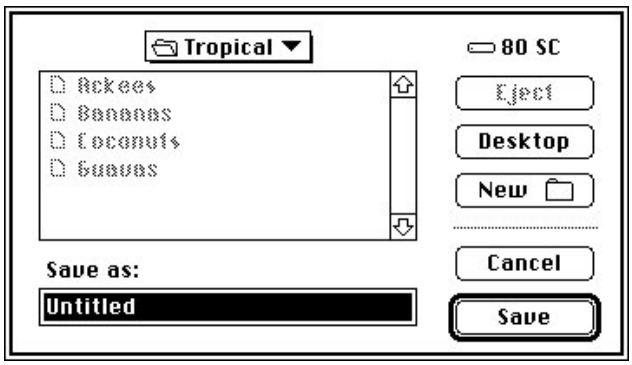
Note that the list of files and folders always contains all folders in the current directory, but it might not contain all files in the current directory. When you call `StandardGetFile`, you can supply a list of the file types that your application can open. The `StandardGetFile` procedure then displays only files of the specified types. You can also supply your own file filter function to help determine which files are displayed. (See "Writing a File Filter Function" on page 3-20 for details.)

When the user is opening a document, `StandardGetFile` interprets some keystrokes as selectors in the displayed list. If the user presses A, for example, `StandardGetFile` selects the first item in the list that starts with the letter *a* (or, if no items in the list start with the letter *a*, the item that starts with the letter closest to *a*). The Standard File Package sets a timer on keystrokes: keystrokes in rapid succession form a string; keystrokes spaced in time are processed separately. See "Keyboard Equivalents" on page 3-7 for a complete list of keyboard equivalents recognized by `StandardGetFile`.

Saving Files

You use the `StandardPutFile` procedure when you want to let the user specify a name and location for a file to be saved. Figure 3-2 illustrates a sample dialog box displayed by `StandardPutFile`.

Figure 3-2 The default Save dialog box

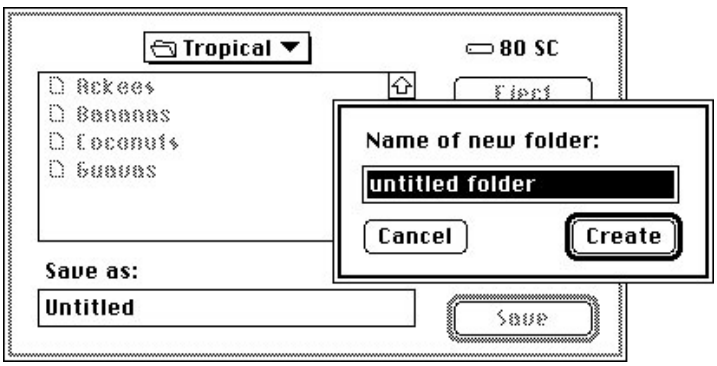


The dialog box displayed by `StandardPutFile` is similar to that displayed by `StandardGetFile`, but includes three additional items. The Save dialog box includes a filename field in which the user can type the name under which to save the file. This filename field is a TextEdit field that permits all the standard editing operations (cut, copy, paste, and so forth). Above the filename field is a line of text specified by your application.

When the user is saving a document, `StandardPutFile` can direct keystrokes to either of two targets: the filename field or the displayed list. When the dialog box first appears, keystrokes are directed to the filename field. If the user presses the Tab key or clicks to select an item in the displayed list, subsequent keystrokes are interpreted as selectors in the displayed list. Each time the user presses the Tab key, keyboard input shifts between the two targets.

The third additional item in the Save dialog is the New Folder button. When the user clicks the New Folder button, the Standard File Package presents a subsidiary dialog box like the one shown in Figure 3-3.

Figure 3-3 The New Folder dialog box



If the user asks to save a file under a name that already exists at the specified location, the Standard File Package displays a subsidiary dialog box to verify that the new file should replace the existing file, as illustrated in Figure 3-4.

Figure 3-4 The name conflict dialog box

The `StandardGetFile` and `StandardPutFile` procedures always display the new dialog boxes. The procedures available before version 7.0 (`SFGetFile`, `SFPutFile`, `SFPGetFile`, and `SFPPutFile`) also display the new dialog boxes when running in version 7.0, unless your application has customized the dialog box. For more details on how the version 7.0 Standard File Package handles earlier procedures, see “Using the Original Procedures” on page 3-40.

Keyboard Equivalents

The Standard File Package recognizes a long list of keyboard equivalents during dialogs.

Keystrokes	Action
Up Arrow	Scroll up (backward) through displayed list
Down Arrow	Scroll down (forward) through displayed list
Command–Up Arrow	Display contents of parent directory
Command–Down Arrow	Display contents of selected directory or volume
Command–Left Arrow	Display contents of previous volume
Command–Right Arrow	Display contents of next volume
Command–Shift–Up Arrow	Display contents of desktop
Command-Shift-1	Eject disk in drive 1
Command-Shift-2	Eject disk in drive 2
Tab	Move to next keyboard target
Return *or* Enter	Invoke the default option for the dialog box (Open or Save)
Escape *or* Command-.	Cancel
Command-O	Open the selected item
Command-D	Display contents of desktop
Command-N	Create a new folder
Option-Command-O *or* Option-[click Open]	Select the target of the selected alias item instead of opening it

When the user uses a keyboard equivalent to select a button in the dialog box, the button blinks.

Customized User Interfaces

The standard user interfaces provided by the `StandardGetFile` and `StandardPutFile` procedures might not be adequate for the needs of certain applications. To handle such cases, the Standard File Package provides several routines that you can use to present a customized user interface when opening or saving files. This section gives some simple examples of how you might want to customize the user interfaces and suggests some guidelines you should follow when doing so.

IMPORTANT

You should alter the standard user interfaces only if necessary. Apple Computer, Inc., does not guarantee future compatibility for your application if you use a customized dialog box. ▲

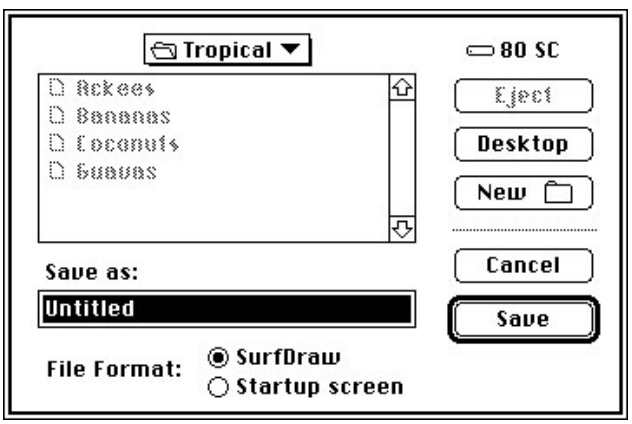
Saving Files

Perhaps the most common reason to customize one of the Standard File Package dialog boxes is to allow the user to save a document in one of several file formats supported by the application. For example, a word-processing application might let the user save a document in the application's own format, in an interchange format, as a file of type `'TEXT'`, and so on.

It is usually best to allow the user to select a file format from within the dialog box displayed in response to a Save or Save As menu command. To do this, you need to add items to the standard dialog box and process user actions in those new items.

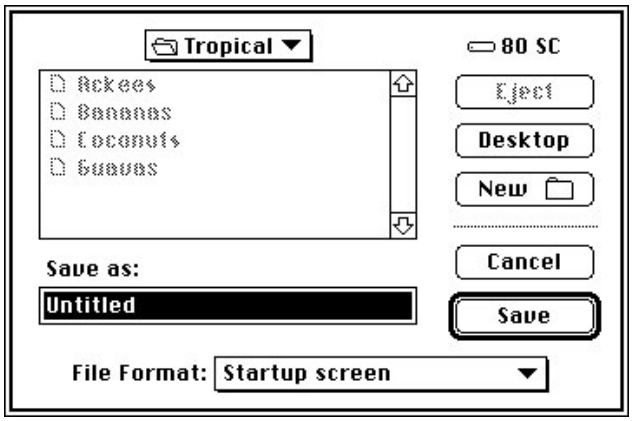
If your application supports only a few file formats, you could simply add the required number of radio buttons to the standard dialog box, as illustrated in Figure 3-5. The application presenting this dialog box supports only two file formats, its own proprietary format (SurfDraw) and the format used for startup screens.

Figure 3-5 The Save dialog box customized with radio buttons



If your application supports more than a couple of alternate file formats, you could add a pop-up menu, as shown in Figure 3-6.

Figure 3-6 The Save dialog box customized with a pop-up menu

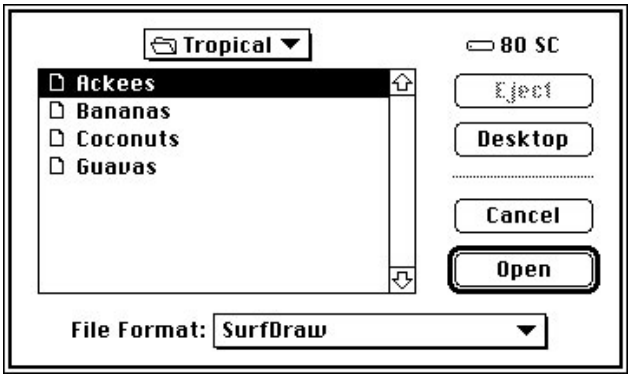


Opening Files

Your application might also allow the user to open a number of different types of files. In this case, there is less need to customize the Open dialog box than the Save dialog box because you can simply list all the kinds of files your application supports. To avoid clutter in the list of files and folders, however, you might wish to filter out all but one of those types. In this way, the user can dynamically select which type of file to view in the list.

Once again, you might accomplish this by adding radio buttons or a pop-up menu to the Open dialog box, depending on the number of different file types your application supports. Figure 3-7 illustrates a customized Open dialog box that contains a pop-up menu. Only files of the indicated type (and, of course, folders) appear in the list of items available to open.

Figure 3-7 The Open dialog box customized with a pop-up menu



For details on some techniques you can use to add items to the standard user interface and process user actions with those additional items, see “Customizing the User Interface” on page 3-16. Note in particular that Listing 3-3, Listing 3-8, and Listing 3-9 together provide a fairly complete implementation of the pop-up menu illustrated in Figure 3-7.

Note

Remember that the user might also open one of your application’s documents from the Finder (by double-clicking its icon, for example). As a result, you should in general avoid customizing the Open dialog box for files. ◆

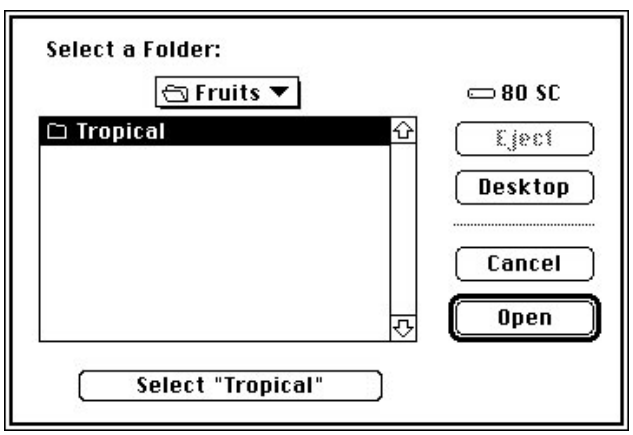
Selecting Volumes and Directories

Sometimes you need to allow the user to select a directory or a volume, not a file. For example, the user might want to select a directory as a first step in searching all the files in the directory for some important information. Similarly, the user might need to select a volume before backing up all the files on that volume.

The standard Open dialog box, however, is designed for selecting files, not volumes or directories. When the user selects a volume or directory from the items in the displayed list and clicks the Open button, the volume or directory is opened and its contents are displayed in the list. The standard Open dialog boxes provide no obvious mechanism for choosing a selected directory instead of opening it.

To allow the user to select a directory—including the volume’s root directory, the volume itself—you can add an additional button to the standard Open dialog box. By clicking this button, the user can select a highlighted directory, not open it. This button gives the user an obvious way to select a directory while preserving the well-known mechanism for opening directories to search for the desired directory. Figure 3-8 illustrates the standard Open dialog box modified to include a Select button and a prompt informing the user of the type of action required.

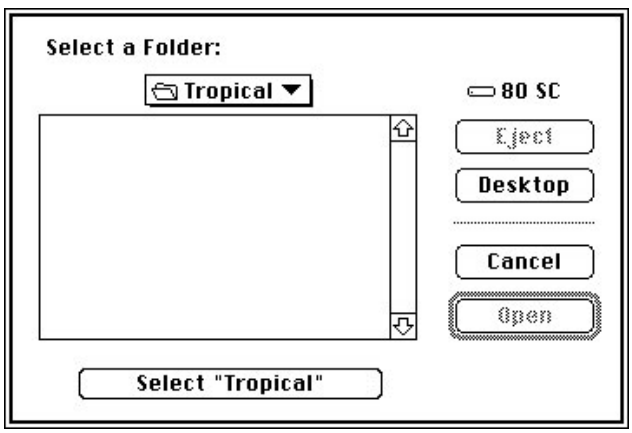
Figure 3-8 The Open dialog box customized to allow selection of a directory



The Select button should display the name of the directory that is selected if the user clicks the button. This, together with the prompt displayed at the top of the dialog box, helps the user differentiate this directory selection dialog box from the standard file opening dialog box. All the other items in the dialog box should maintain their standard appearance and behavior. Any existing keyboard equivalents (in particular, the use of Return and Enter to select the default button) should be preserved. Command-S is recommended as a keyboard equivalent for the new Select button, paralleling the use of Command-D to select the Desktop button and Command-O to select the Open button.

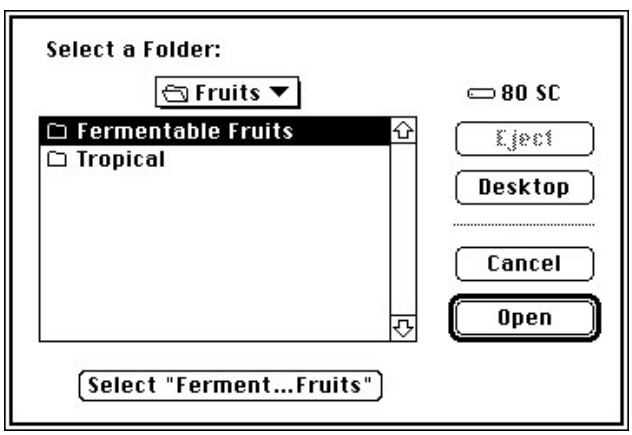
To help maintain consistency among applications using this scheme for selecting directories, your application should open the folder displayed in the pop-up menu if there is no selected item and the user clicks the Select button. In addition, you should disable the Open button if no directory is currently selected. Figure 3-9 illustrates the recommended appearance of the directory selection dialog box in this case.

Figure 3-9 The Open dialog box when no directory is selected



If the name of the directory is too long to fit in the Select button, you should abbreviate the name using an ellipsis character, as shown in Figure 3-10.

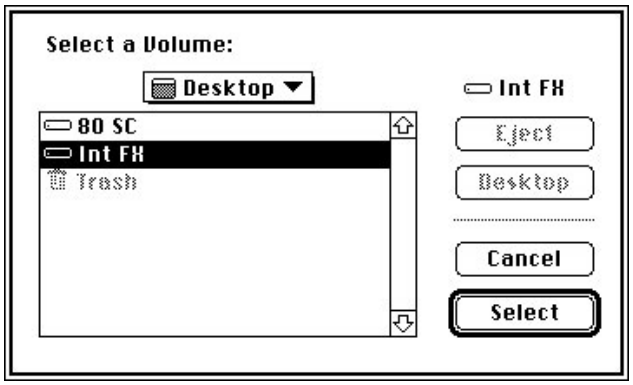
Figure 3-10 The Open dialog box with a long directory name abbreviated



See “Selecting a Directory” beginning on page 3-34 for details on how you can create and manage a directory selection dialog box.

The directory selection dialog boxes illustrated here allow the user to specify the root directory in a volume, which effectively selects the volume itself. However, you might want to limit the user’s selections to the available volumes. To do that, you can create a volume selection dialog box, shown in Figure 3-11.

Figure 3-11 A volume selection dialog box



Notice that the volume selection dialog box uses a prompt specific to selecting a volume and that the Open button is now a Select button. There is no need for a separate Select button, because the user should not be allowed to open any of the listed volumes. (For this same reason, the pop-up menu should not pop up if clicked.) See “Selecting a Volume” on page 3-38 for instructions on implementing a volume selection dialog box.

User Interface Guidelines

In general, you should customize the user interface only if necessary. If you do modify the standard dialog boxes presented by the Standard File Package, you should keep these user interface guidelines in mind:

- Customize a dialog box only by adding items to the standard dialog boxes. Avoid removing existing items from the standard boxes or altering the operation of existing items. In particular, you should avoid modifying the keyboard equivalents recognized by the Standard File Package.
- Add only those items that are necessary for your application to complete the requested action successfully. Avoid adding items that provide unnecessary information or items that provide no information at all (such as logos, icons, or other “window-dressing”).
- Whenever possible, use controls such as radio buttons or pop-up menus whose effects are visible within the dialog box itself. Avoid controls whose use calls subsidiary modal dialog boxes that the user must dismiss before continuing.
- Use controls or other items that are already familiar to the user. Avoid using customized controls that are not also used elsewhere in your application.

Your overriding concern should be to make the customized file identification dialog boxes in your application as similar to the standard dialog boxes as possible while providing the additional capabilities you need.

Using the Standard File Package

You use the Standard File Package to handle the user interface when the user must specify a file to be saved or opened. You typically call the Standard File Package after the user chooses Save, Save As, or Open from the File menu.

When saving a document, you call one of the `PutFile` procedures; when opening a document, you call one of the `GetFile` procedures. The Standard File Package in version 7.0 introduces two pairs of enhanced procedures:

- `StandardPutFile` and `StandardGetFile`, for presenting the standard interface
- `CustomPutFile` and `CustomGetFile`, for presenting a customized interface

Before calling the enhanced Standard File Package procedures, verify that they are available by calling the `Gestalt` function with the `gestaltStandardFileAttr` selector. If `Gestalt` sets the `gestaltStandardFile58` bit in the reply, the four enhanced procedures are available.

If the enhanced procedures are not available, you need to use the original Standard File Package procedures that are available in all system software versions:

- `SFPutFile` and `SFGetFile`, for presenting the standard interface
- `SFPPutFile` and `SFPGetFile`, for presenting a customized interface

This section focuses on the enhanced procedures introduced in system software version 7.0. If you need to use the original procedures, see “Using the Original Procedures” on page 3-40. You can adapt most of the techniques shown in this section for use with the original procedures. In general, however, the original procedures are slightly harder to use and somewhat less powerful than their enhanced counterparts.

All the enhanced procedures return the results of the dialog boxes in a new reply record, `StandardFileReply`.

```
TYPE StandardFileReply =
   RECORD
      sfGood:        Boolean;    {TRUE if user did not cancel}
      sfReplacing:   Boolean;    {TRUE if replacing file with same name}
      sfType:        OSType;     {file type}
      sfFile:        FSSpec;     {selected file, folder, or volume}
      sfScript:      ScriptCode; {script of file, folder, or volume name}
      sfFlags:       Integer;    {Finder flags of selected item}
      sfIsFolder:    Boolean;    {selected item is a folder}
      sfIsVolume:    Boolean;    {selected item is a volume}
      sfReserved1:   LongInt;    {reserved}
      sfReserved2:   Integer;    {reserved}
   END;
```

The reply record identifies selected files with a file system specification (FSSpec) record. You can pass the FSSpec record directly to the File Manager functions that recognize FSSpec records, such as FSpOpenDF or FSpCreate. The reply record also contains additional fields that support the Finder features introduced in system software version 7.0.

The sfGood field reports whether the reply record is valid—that is, whether your application can use the information in the other fields. The field is set to TRUE after the user clicks Save or Open, and to FALSE after the user clicks Cancel.

Your application needs to look primarily at the sfFile and sfReplacing fields when the sfGood field contains TRUE. The sfFile field contains a file system specification record that describes the selected file or folder. If the selected file is a stationery pad, the reply record describes the file itself, not a copy of the file.

The sfReplacing field reports whether a file to be saved replaces an existing file of the same name. This field is valid only after a call to the StandardPutFile or CustomPutFile procedure. Your application can rely on the value of this field instead of checking for and handling name conflicts itself.

Note

See "Enhanced Standard File Reply Record" on page 3-42 for a complete description of the fields of the StandardFileReply record. ◆

The Standard File Package fills in the reply record and returns when the user completes one of its dialog boxes—either by selecting a file and clicking Save or Open, or by clicking Cancel. Your application checks the values in the reply record to see what action to take, if any. If the selected item is an alias for another item, the Standard File Package resolves the alias and places a file system specification record for the target in the sfFile field when the user completes the dialog box. (See the chapter "Finder Interface" of *Inside Macintosh: Macintosh Toolbox Essentials* for a description of aliases.)

Presenting the Standard User Interface

You can use the standard dialog boxes provided by the Standard File Package to prompt the user for the name of a file to open or a filename and location to use when saving a document. Use StandardGetFile to present the standard interface when opening a file and StandardPutFile to present the standard interface when saving a file.

Listing 3-1 illustrates how your application can use StandardGetFile to elicit a file specification after the user chooses Open from the File menu.

Listing 3-1 Handling the Open menu command

```
FUNCTION DoOpenCmd: OSErr;
VAR
   myReply:    StandardFileReply;   {Standard File reply record}
   myTypes:    SFTypeList;          {types of files to display}
   myErr:      OSErr;
```

```
BEGIN
   myTypes[0] := 'TEXT';            {display text files only}
   StandardGetFile(NIL, 1, myTypes, myReply);
   IF myReply.sfGood THEN
      myErr := DoOpenFile(myReply.sfFile)
   ELSE
      myErr := UsrCanceledErr;
   DoOpenCmd := myErr;
END;
```

If the user dismisses the dialog box by clicking the Open button, the reply record field `myReply.sfGood` is set to `TRUE`; in that case, the function defined in Listing 3-1 calls the application-defined function `DoOpenFile`, passing it the file system specification record contained in the reply record. For a sample definition of the `DoOpenFile` function, see the chapter "Introduction to File Management" in this book.

The third parameter to `StandardGetFile` is a list of file types that are to appear in the list of files and folders; the second parameter is the number of items in that list of file types. The list of file types is of type `SFTypeList`.

```
TYPE  SFTypeList  =  ARRAY[0..3] OF OSType;
```

If you need to display more than four types of files, you can define a new data type that is large enough to hold all the types you need. For example, you can define the data type `MyTypeList` to hold ten file types:

```
TYPE  MyTypeList  =  ARRAY[0..9] OF OSType;
      MyTListPtr  =  ^MyTypeList;
```

Listing 3-2 shows how to call `StandardGetFile` using an expanded type list.

Listing 3-2 Specifying more than four file types

```
FUNCTION DoOpenCmd: OSErr;
VAR
   myReply:    StandardFileReply;   {Standard File reply record}
   myTypes:    MyTypeList;          {types of files to display}
   myErr:      OSErr;
BEGIN
   myTypes[0] := 'TEXT';            {first file type to display}
   {Put other assignments here.}
   myTypes[9] := 'RTFT';            {tenth file type to display}
   StandardGetFile(NIL, 1, MyTListPtr(myTypes)^, myReply);
   IF myReply.sfGood THEN
      myErr := DoOpenFile(myReply.sfFile)
```

```
   ELSE
      myErr := UsrCanceledErr;
   DoOpenCmd := myErr;
END;
```

Note

To display all file types in the dialog box, pass –1 as the second parameter. Invisible files and folders are not shown in the dialog box unless you pass –1 in that parameter. If you pass –1 as the second parameter when calling `CustomGetFile`, the dialog box also lists folders; this is not true when you call `StandardGetFile`. ◆

The first parameter passed to `StandardGetFile` is the address of a file filter function, a function that helps determine which files appear in the list of files to open. (In Listing 3-1, this address is `NIL`, indicating that all files of the specified type are to be listed.) See "Writing a File Filter Function" on page 3-20 for details on defining a filter function for use with `StandardGetFile`.

Customizing the User Interface

If your application requires it, you can customize the user interface for identifying files. To customize a dialog box, you should

- design your dialog box and create the resources that describe it
- write callback routines, if necessary, to process user actions in the dialog box
- call the Standard File Package using the `CustomPutFile` and `CustomGetFile` procedures, passing the resource IDs of the customized dialog boxes and pointers to the callback routines

Depending on the level of customizing you require in your dialog box, you may need to write as many as four different callback routines:

- a file filter function for determining which files the user can open
- a dialog hook function for handling user actions in the dialog boxes
- a modal-dialog filter function for handling user events received from the Event Manager
- an activation procedure for highlighting the display when keyboard input is directed at a customized field defined by your application

To provide the interface illustrated in Figure 3-7, for example, you could replace the definition of `DoOpenCmd` given earlier in Listing 3-1 by the definition given in Listing 3-3.

In addition to the information passed to `StandardGetFile`, `CustomGetFile` requires the resource ID of the customized dialog box, the location of the dialog box on the screen, and pointers to any callback routines and private data you are using.

Listing 3-3 Presenting a customized Open dialog box

```
FUNCTION DoOpenCmd: OSErr;
VAR
   myReply:    StandardFileReply;   {Standard File reply record}
   myTypes:    SFTypeList;          {types of files to display}
   myPoint:    Point;               {upper-left corner of box}
   myErr:      OSErr;
CONST
   kCustomGetDialog = 4000;         {resource ID of custom dialog}
BEGIN
   myErr := noErr;
   SetPt(myPoint, -1, -1);          {center the dialog}
   myTypes[0] := 'SRFD';            {SurfDraw files}
   myTypes[1] := 'STUP';            {startup screens}
   myTypes[2] := 'PICT';            {picture files}
   myTypes[3] := 'RTFT';            {rich text format}

   CustomGetFile(@MyCustomFileFilter, 4, myTypes, myReply,
                 kCustomGetDialog, myPoint, @MyDlgHook,
                 NIL, NIL, NIL, NIL);
   IF myReply.sfGood THEN
      myErr := DoOpenFile(myReply.sfFile);
   DoOpenCmd := myErr;
END;
```

In Listing 3-3, `CustomGetFile` is passed two callback routines, a file filter function (`MyCustomFileFilter`) and a dialog hook function (`MyDlgHook`). See Listing 3-8 (page 3-21) and Listing 3-9 (page 3-27) for sample definitions of these functions.

You can also supply data of your own to the callback routines through a new parameter, `yourDataPtr`, which you pass to `CustomGetFile` and `CustomPutFile`.

Customizing Dialog Boxes

To describe a dialog box, you supply a `'DLOG'` resource that defines the box itself and a `'DITL'` resource that defines the items in the dialog box.

Listing 3-4 shows the resource definition of the default Open dialog box, in Rez input format. (Rez is the resource compiler provided with Apple's Macintosh Programmer's Workshop [MPW]. For a description of Rez format, see the manual that accompanies the MPW software, *MPW: Macintosh Programmer's Development Environment*.)

Listing 3-4 The definition of the default Open dialog box

```
resource 'DLOG' (-6042, purgeable)
   {
      {0, 0, 166, 344}, dBoxProc, invisible, noGoAway, 0,
       –6042, "", noAutoCenter
   };
```

Listing 3-5 shows the resource definition of the default Save dialog box, in Rez input format.

Listing 3-5 The definition of the default Save dialog box

```
resource 'DLOG' (-6043, purgeable)
   {
      {0, 0, 188, 344}, dBoxProc, invisible, noGoAway, 0,
       –6043, "", noAutoCenter
   };
```

Note
You can also use the stand-alone resource editor ResEdit, available from Apple Computer, Inc., or other resource-editing utilities available from third-party developers to create customized dialog box and dialog item list resources. ◆

You must provide an item list (in a `'DITL'` resource with the ID specified in the `'DLOG'` resource) for each dialog box you define. Add new items to the end of the default lists. `CustomGetFile` expects the first 9 items in a customized dialog box to have the same functions as the corresponding items in the `StandardGetFile` dialog box; `CustomPutFile` expects the first 12 items to have the same functions as the corresponding items in the `StandardPutFile` dialog box. If you want to eliminate one of the standard items from the display, leave it in the item list but place its coordinates outside the bounds of the dialog box rectangle.

Listing 3-6 shows the dialog item list for the default Open dialog box, in Rez input format. See “Writing a Dialog Hook Function” beginning on page 3-21 for a list of the dialog box elements these items represent.

Listing 3-6 The item list for the default Open dialog box

```
resource 'DITL'(-6042)
   {  {
      {135, 252, 155, 332}, Button { enabled, "Open" },
      {104, 252, 124, 332}, Button { enabled, "Cancel" },
      {0, 0, 0, 0}, HelpItem { disabled, HMScanhdlg {-6042}},
      {8, 235, 24, 337}, UserItem { enabled },
```

```
      {32, 252, 52, 332}, Button { enabled, "Eject" },
      {60, 252, 80, 332}, Button { enabled, "Desktop" },
      {29, 12, 159, 230}, UserItem { enabled },
      {6, 12, 25, 230}, UserItem { enabled },
      {91, 251, 92, 333}, Picture  { disabled, 11 },
   }  };
```

Listing 3-7 shows the dialog item list for the default Save dialog box, in Rez input format.

Listing 3-7 The item list for the default Save dialog box

```
resource 'DITL'(-6043)
   {  {
      {161, 252, 181, 332}, Button { enabled, "Save" },
      {130, 252, 150, 332}, Button { enabled, "Cancel" },
      {0, 0, 0, 0}, HelpItem { disabled, HMScanhdlg {-6043}},
      {8, 235, 24, 337}, UserItem { enabled },
      {32, 252, 52, 332}, Button { enabled, "Eject" },
      {60, 252, 80, 332}, Button { enabled, "Desktop" },
      {29, 12, 127, 230}, UserItem { enabled },
      {6, 12, 25, 230}, UserItem { enabled },
      {119, 250, 120, 334}, Picture { disabled, 11 },
      {157, 15, 173, 227}, EditText { enabled, "" },
      {136, 15, 152, 227}, StaticText { disabled, "Save as:" },
      {88, 252, 108, 332}, UserItem { disabled },
   }  };
```

The third item in each list (`HelpItem`) supplies Apple's Balloon Help for items in the dialog box. This third item specifies the resource ID of the `'hdlg'` resource that contains the help strings for the standard dialog items. If you want to modify the help text of an existing dialog item, you should copy the original `'hdlg'` resource from the System file into your application's resource fork and modify the text in the copied resource as desired; then you must change the resource ID specified in `HelpItem` to the resource ID of the copied and modified resource. To provide Balloon Help for your own items, supply a second `'hdlg'` resource and reference it with another help item at the end of the list. The existing items retain their default text (unless you change that text, as described).

The default dialog item lists used by the original Standard File Package routines do not contain help items, but the Standard File Package does provide Balloon Help when you call those routines in system software version 7.0 and later. If you call one of the original routines and the specified dialog item list does not contain any help items, the Standard File Package uses its default help text for the standard dialog items. If, however, the dialog item list does contain a help item, the Standard File Package assumes that your application provides the text for all help items, including the standard dialog items.

Note
The default Standard File Package dialog boxes support color. The System file contains `'dctb'` resources with the same resource IDs as the default dialog boxes, so that the Dialog Manager uses color graphics ports for the default dialog boxes. (See the chapter "Dialog Manager" of *Inside Macintosh: Macintosh Toolbox Essentials* for a description of the `'dctb'` resource.) If you create your own dialog boxes, include `'dctb'` resources. ◆

Writing a File Filter Function

A **file filter function** determines which files appear in the displayed list when the user is opening a file. Both `StandardGetFile` and `CustomGetFile` recognize file filter functions.

When the Standard File Package is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application's specifications. (Your application can specify which file types are to be displayed through the `typeList` parameter to either `StandardGetFile` or `CustomGetFile`, as described in "Presenting the Standard User Interface" beginning on page 3-14.) If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

The file filter function receives a pointer to the file's catalog information record (described in the chapter "File Manager" in this book). The function evaluates the catalog entry and returns a Boolean value that determines whether the file is filtered (that is, a value of `TRUE` suppresses display of the filename, and a value of `FALSE` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

A file filter function to be called by `StandardGetFile` must use this syntax:

```
FUNCTION MyStandardFileFilter (pb: CInfoPBPtr): Boolean;
```

The single parameter passed to your standard file filter function is the address of a catalog information parameter block; see the chapter "File Manager" in this book for a description of the fields of that parameter block.

When `CustomGetFile` calls your file filter function, it can also receive a pointer to any data that you passed in through the call to `CustomGetFile`. A file filter function to be called by `CustomGetFile` must use this syntax:

```
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr):
                                    Boolean;
```

Listing 3-8 shows a sample file filter function to be called by `CustomGetFile`. You might define a file filter function like this to support the custom dialog box illustrated in Figure 3-7, which lists files of the type shown in the pop-up box.

Listing 3-8 A sample file filter function

```
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr): Boolean;
BEGIN
   MyCustomFileFilter := TRUE;              {default: don't show the file}
   IF pb^.ioFlFndrInfo.fdType = gTypesArray[gCurrentType] THEN
      MyCustomFileFilter := FALSE;          {show the file}
END;
```

In Listing 3-8, the application global variable `gCurrentType` contains the index in the array `gTypesArray` of the currently selected file type. If the type of a file passed in for evaluation matches the current file type, the filter returns `FALSE`, indicating that `StandardGetFile` should put it in the list. See Listing 3-9 (page 3-27) for an example of how you can use a dialog hook function to change the value of `gCurrentType` according to user selections in the pop-up menu control.

Writing a Dialog Hook Function

A **dialog hook function** handles item selections in a dialog box. It receives a pointer to the dialog record and an item number from the `ModalDialog` procedure via the Standard File Package each time the user selects one of the dialog items. Your dialog hook function checks the item number of each selected item, and then either handles the selection or passes it back to the Standard File Package.

If you provide a dialog hook function, `CustomPutFile` and `CustomGetFile` call your function immediately after calling `ModalDialog`. They pass your function the item number returned by `ModalDialog`, a pointer to the dialog record, and a pointer to the data received from your application, if any. The dialog hook function must use this syntax:

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr;
                    myDataPtr: Ptr): Integer;
```

Your dialog hook function returns as its function result an integer that is either the item number passed to it or some other item number. If it returns one of the item numbers in the following list of constants, the Standard File Package handles the selected item as described later in this section. If your dialog hook function does not handle a selection, it should pass the item number back to the Standard File Package for processing by setting its return value equal to the item number.

```
CONST {items that appear in both the Open and Save dialog boxes}
   sfItemOpenButton        =  1;       {Save or Open button}
   sfItemCancelButton      =  2;       {Cancel button}
   sfItemBalloonHelp       =  3;       {Balloon Help}
   sfItemVolumeUser        =  4;       {volume icon and name}
   sfItemEjectButton       =  5;       {Eject button}
   sfItemDesktopButton     =  6;       {Desktop button}
   sfItemFileListUser      =  7;       {display list}
```

```
   sfItemPopUpMenuUser     =  8;       {directory pop-up menu}
   sfItemDividerLinePict   =  9;       {dividing line between buttons}

   {items that appear in Save dialog boxes only}
   sfItemFileNameTextEdit  =  10;      {filename field}
   sfItemPromptStaticText  =  11;      {filename prompt text area}
   sfItemNewFolderUser     =  12;      {New Folder button}
```

You must write your own dialog hook function to handle any items you have added to the dialog box.

Note

The constants that represent disabled items (`sfItemBalloonHelp`, `sfItemDividerLinePict`, and `sfItemPromptStaticText`) have no effect, but they are defined in the header files for the sake of completeness. ◆

The Standard File Package also recognizes a number of constants that do not represent any actual item in the dialog list; these constants are known as **pseudo-items.** There are two kinds of pseudo-items:

- pseudo-items passed to your dialog hook function by the Standard File Package
- pseudo-items passed to the Standard File Package by your dialog hook function

The `sfHookFirstCall` constant is an example of the first kind of pseudo-item. The Standard File Package sends this pseudo-item to your dialog hook function immediately before it displays the dialog box. Your function typically reacts to this item number by performing any necessary initialization.

You can pass back other pseudo-items to indicate that you've handled the user selection or to request some action by the Standard File Package. For example, if the list of files and folders must be rebuilt because of a user selection, you can pass back the pseudo-item `sfHookRebuildList`. Similarly, when your application handles the selection and needs no further action by the Standard File Package, it should return `sfHookNullEvent`. When the dialog hook function passes either `sfHookNullEvent` or an item number that the Standard File Package doesn't recognize, it does nothing.

The Standard File Package recognizes these pseudo-item numbers:

```
CONST {pseudo-items available prior to version 7.0}
   sfHookFirstCall         =  -1;      {initialize display}
   sfHookCharOffset        =  $1000;   {offset for character input}
   sfHookNullEvent         =  100;     {null event}
   sfHookRebuildList       =  101;     {redisplay list}
   sfHookFolderPopUp       =  102;     {display parent-directory menu}
   sfHookOpenFolder        =  103;     {display contents of }
                                       { selected folder or volume}

   {additional pseudo-items introduced in version 7.0}
   sfHookLastCall          =  -2;      {clean up after display}
```

```
sfHookOpenAlias         =  104;     {resolve alias}
sfHookGoToDesktop       =  105;     {display contents of desktop}
sfHookGoToAliasTarget   =  106;     {select target of alias}
sfHookGoToParent        =  107;     {display contents of parent}
sfHookGoToNextDrive     =  108;     {display contents of next drive}
sfHookGoToPrevDrive     =  109;     {display contents of previous drive}
sfHookChangeSelection   =  110;     {select target of reply record}
sfHookSetActiveOffset   =  200;     {switch active item}
```

The Standard File Package uses a set of modal-dialog filter functions (described in "Writing a Modal-Dialog Filter Function" on page 3-28) to map user actions during the dialog onto the defined item numbers. Some of the mapping is indirect. A click of the Open button, for example, is mapped to `sfItemOpenButton` only if a file is selected in the display list. If a folder or volume is selected, the Standard File Package maps the selection onto the pseudo-item `sfHookOpenFolder`.

The lists that follow summarize when various items and pseudo-items are generated and how they are handled. The descriptions indicate the simplest mouse action that generates each item; many of the items can also be generated by keyboard actions, as described in "Keyboard Equivalents" on page 3-7.

Note

Any indicated effects of passing back these constants do not occur until the Standard File Package receives the constant back from your dialog hook function. ◆

Constant descriptions

`sfItemOpenButton`
: Generated when the user clicks Open or Save while a filename is selected. The Standard File Package fills in the reply record (setting `sfGood` to `TRUE`), removes the dialog box, and returns.

`sfItemCancelButton`
: Generated when the user clicks Cancel. The Standard File Package sets `sfGood` to `FALSE`, removes the dialog box, and returns.

`sfItemVolumeUser`
: Generated when the user clicks the volume icon or its name. The Standard File Package rebuilds the display list to show the contents of the folder that is one level up the hierarchy (that is, the parent directory of the current parent directory).

`sfItemEjectButton`
: Generated when the user clicks Eject. The Standard File Package ejects the volume that is currently selected.

`sfItemDesktopButton`
: Generated when the user clicks the Drive button in a customized dialog box defined by one of the earlier procedures. You never receive this item number with the new procedures; when the user clicks the Desktop button, the action is mapped to the item `sfHookGoToDesktop`, described later in this section. The Standard File Package displays the contents of the next drive.

`sfItemFileListUser`
: Generated when the user clicks an item in the display list. The Standard File Package updates the selection and generates this item for your information.

`sfItemPopUpMenuUser`
: Never generated. The Standard File Package's modal-dialog filter function maps clicks on the directory pop-up menu to `sfHookFolderPopUp`, described later in this section.

`sfItemFileNameTextEdit`
: Generated when the user clicks the filename field. TextEdit and the Standard File Package process mouse clicks in the filename field, but the item number is generated for your information.

`sfItemNewFolderUser`
: Generated when the user clicks New Folder. The Standard File Package displays the New Folder dialog box.

The pseudo-items are messages that allow your application and the Standard File Package to communicate and support various features added since the original design of the Standard File Package.

The Standard File Package generates three pseudo-items that give your application the chance to control a customized display.

Constant descriptions

`sfHookFirstCall`
: Generated by the Standard File Package as a signal to your dialog hook function that it is about to display a dialog box. If you want to initialize the display, do so when you receive this item. You can specify the current directory either by returning `sfHookGoToDesktop` or by changing the reply record and returning `sfHookChangeSelection`.

`sfHookLastCall` Generated by the Standard File Package as a signal to your dialog hook function that it is about to remove a dialog box. If you created any structures when the dialog box was first displayed, remove them when you receive this item.

`sfHookNullEvent`
: Issued periodically by the Standard File Package if no user action has taken place. Your application can use this null event to perform any updating or periodic processing that might be necessary.

Your application can generate three pseudo-items to request services from the Standard File Package.

Constant descriptions

`sfHookRebuildList`
: Returned by your dialog hook function to the Standard File Package when it needs to redisplay the file list. Your application might need to redisplay the list if, for example, it allows the user to change the file types to be displayed. The Standard File Package rebuilds and displays the list of files that can be opened.

`sfHookChangeSelection`

Returned by your application to the Standard File Package after your application changes the reply record so that it describes a different file or folder. (You'll need to pass the address of the reply record in the `yourDataPtr` field if you want to do this.) The Standard File Package rebuilds the display list to show the contents of the folder or volume containing the object described in the reply record. It selects the item described in the reply record.

`sfHookSetActiveOffset`

Your application adds this constant to an item number and sends the result to the Standard File Package. The Standard File Package activates that item in the dialog box, making it the target of keyboard input. This constant allows your application to activate a specific field in the dialog box without explicit input from the user.

The Standard File Package's own modal-dialog filter functions generate a number of pseudo-items that allow its dialog hook functions to support various features introduced since the original design of the standard file dialog boxes. Except under extraordinary circumstances, your dialog hook function always passes any of these item numbers back to the Standard File Package for processing.

Constant descriptions

`sfHookCharOffset`

The Standard File Package adds this constant to the value of an ASCII character when it's using keyboard input for item selection. The Standard File Package uses the decoded ASCII character to select an entry in the display list.

`sfHookFolderPopUp`

Generated when the user clicks the directory pop-up menu. The Standard File Package displays the pop-up menu showing all parent directories.

`sfHookOpenFolder`

Generated when the user clicks the Open button while a folder or volume is selected in the display list. The Standard File Package rebuilds the display list to show the contents of the folder or volume.

`sfHookOpenAlias`

Generated by the Standard File Package as a signal that the selected item is an alias for another file, folder, or volume. If the selected item is an alias for a file, the Standard File Package resolves the alias, places the file system specification record of the target in the reply record, and returns.

If the selected item is an alias for a folder or volume, the Standard File Package resolves the alias and rebuilds the display list to show the contents of the alias target.

`sfHookGoToDesktop`

Generated when the user clicks the Desktop button. The Standard File Package displays the contents of the desktop in the display list.

`sfHookGoToAliasTarget`
: Generated when the user presses the Option key while opening an item that is an alias. The Standard File Package rebuilds the display list to display the volume or folder containing the alias target and selects the target.

`sfHookGoToParent`
: Generated when the user presses Command–Up Arrow (or clicks the volume icon). The Standard File Package rebuilds the display list to show the contents of the folder that is one level up the hierarchy (that is, the parent directory of the current parent directory).

`sfHookGoToNextDrive`
: Generated when the user presses Command–Right Arrow. The Standard File Package displays the contents of the next volume.

`sfHookGoToPrevDrive`
: Generated when the user presses Command–Left Arrow. The Standard File Package displays the contents of the previous volume.

The `CustomGetFile` and `CustomPutFile` procedures call your dialog hook function for item selections in both the main dialog box and any subsidiary dialog boxes (such as the dialog box for naming a new folder while saving a document through `CustomPutFile`). To determine whether the dialog record describes the main dialog box or a subsidiary dialog box, check the value of the `refCon` field in the window record in the dialog record.

Note

Prior to system software version 7.0, the Standard File Package did not call your dialog hook function during subsidiary dialog boxes. Dialog hook functions for the new `CustomGetFile` and `CustomPutFile` procedures must check the dialog window's `refCon` field to determine the target of the dialog record. ◆

The defined values for the `refCon` field represent the Standard File dialog boxes.

```
CONST
   sfMainDialogRefCon      =  'stdf';  {main dialog box}
   sfNewFolderDialogRefCon =  'nfdr';  {New Folder dialog box}
   sfReplaceDialogRefCon   =  'rplc';  {name conflict dialog box}
   sfStatWarnDialogRefCon  =  'stat';  {stationery warning}
   sfErrorDialogRefCon     =  'err ';  {general error report}
   sfLockWarnDialogRefCon  =  'lock';  {software lock warning}
```

Constant descriptions

`sfMainDialogRefCon` The main dialog box, either Open or Save.

`sfNewFolderDialogRefCon` The New Folder dialog box.

`sfReplaceDialogRefCon` The dialog box requesting verification for replacing a file of the same name.

`sfStatWarnDialogRefCon` The dialog box warning that the user is opening the master copy of a stationery pad, not a piece of stationery.

`sfErrorDialogRefCon`	A dialog box reporting a general error.
`sfLockWarnDialogRefCon`	The dialog box warning that the user is opening a locked file and won't be allowed to save any changes.

Listing 3-9 defines a dialog hook function that handles user selections in the customized Open dialog box illustrated in Figure 3-7. Note that this dialog hook function handles selections only in the main dialog box, not in any subsidiary dialog boxes.

Listing 3-9 A sample dialog hook function

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
                  Integer;
VAR
   myType:        Integer;       {menu item selected}
   myHandle:      Handle;        {needed for GetDItem}
   myRect:        Rect;          {needed for GetDItem}
   myIgnore:      Integer;       {needed for GetDItem; ignored}
CONST
   kMyPopUpItem = 10;            {item number of File Type pop-up menu}
BEGIN
   MyDlgHook := item;            {by default, return the item passed in}
   IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
      Exit(MyDlgHook);           {this function is only for main dialog}

   {Do processing of pseudo-items and your own additional item.}
   CASE item OF
      sfHookFirstCall:           {pseudo-item: first time function called}
         BEGIN
            GetDItem(theDialog, kPopUpItem, myType, myHandle, myRect);
            SetCtlValue(ControlHandle(myHandle), gCurrentType);
            MyDlgHook := sfHookNullEvent;
         END;
      kMyPopUpItem:              {user selected File Type pop-up menu}
         BEGIN
            GetDItem(theDialog, item, myIgnore, myHandle, myRect);
            myType := GetCtlValue(ControlHandle(myHandle));
            IF myType <> gCurrentType THEN
               BEGIN
                  gCurrentType := myType;
                  MyDlgHook := sfHookRebuildList;
               END;
         END;
      OTHERWISE
         ;                       {ignore all other items}
   END;
END;
```

The pop-up menu is stored as a control in the application's resource fork. Values stored in the resource determine the appearance of the control, such as the pop-up title text and the menu associated with the control. The Dialog Manager's `ModalDialog` procedure takes care of drawing the box around the pop-up menu and the title of the dialog box. When the dialog hook function is first called, it simply retrieves a handle to that control and sets the value of the pop-up control to the current menu item (stored in the global variable `gCurrentType`). The `MyDlgHook` function then returns `sfHookNullEvent` to indicate that no further processing is required.

When the user clicks the pop-up menu control, `ModalDialog` calls the standard control definition function associated with it. If the user makes a selection in the pop-up menu, `MyDlgHook` is called with the `item` parameter equal to `kPopUpItem`. Your dialog hook function needs simply to determine the current value of the control and respond accordingly. In this case, if the user has selected a new file type, the global variable `gCurrentType` is updated to reflect the new selection, and `MyDlgHook` returns `sfHookRebuildList` to cause the Standard File Package to rebuild the list of files and folders displayed in the dialog box.

For complete details on handling pop-up menus, see the chapters "Control Manager" and "Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Writing a Modal-Dialog Filter Function

A **modal-dialog filter function** controls events closer to their source by filtering the events received from the Event Manager. The Standard File Package itself contains an internal modal-dialog filter function that maps keypresses and other user input onto the equivalent dialog box items. If you also want to process events at this level, you can supply your own filter function.

Note
You can supply a modal-dialog filter function only when you use one of the procedures that displays a customized dialog box (that is, `CustomGetFile`, `CustomPutFile`, `SFPGetFile`, or `SFPPutFile`). ◆

Your modal-dialog filter function determines how the Dialog Manager procedure `ModalDialog` filters events. The `ModalDialog` procedure retrieves events by calling the Event Manager function `GetNextEvent`. As just indicated, the Standard File Package contains an internal filter function that performs some preliminary processing on each event it receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

You might provide a modal-dialog filter function for several reasons. If you have customized the Open or Save dialog boxes by adding one or more items, you might want to map some of the user's keypresses to those items in the same way that the internal filter function maps certain keypresses to existing items.

Another reason to provide a modal-dialog filter function is to avoid a problem that can arise if an update event is received for one of your application's windows while a Standard File Package dialog box is displayed.

Note
The problem described in the following paragraph occurs only in system software versions earlier than version 7.0. The internal modal-dialog filter function installed by the Standard File Package when running in version 7.0 and later avoids the problem by passing the update event to your dialog filter and, if your filter doesn't handle the event, mapping it to a null event. ◆

When `ModalDialog` calls `GetNextEvent` and receives the update event, `ModalDialog` does not know how to respond to it and therefore passes the update event to the Standard File Package's internal filter function. The internal filter function cannot handle the update event either. As a result, if you do not provide your own modal-dialog filter function that handles the update event, that event is never cleared. The next time `ModalDialog` calls `GetNextEvent`, it receives the same update event. `ModalDialog` never receives a null event, so your dialog hook function never performs any processing in response to the `sfHookNullEvent` pseudo-item. You can solve this problem by providing a modal-dialog filter function that handles the update event or changes it to a null event. See Listing 3-10 for details.

A modal-dialog filter function used with `SFPGetFile` and `SFPPutFile` is declared like any filter function passed to `ModalDialog`. Your function is passed a pointer to the dialog record, a pointer to the event record, and the item number. (The modal-dialog filter function is described in the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.)

```
FUNCTION MyModalFilter (theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
```

The modal-dialog filter function used with `CustomGetFile` and `CustomPutFile` requires an additional parameter, a pointer (`myDataPtr`) to the data received from your application, if any.

```
FUNCTION MyModalFilterYD (theDialog: DialogPtr;
                          VAR theEvent: EventRecord;
                          VAR itemHit: Integer;
                          myDataPtr: Ptr): Boolean;
```

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `FALSE`, `ModalDialog` processes the event through its own filters. If your function returns a value of `TRUE`, `ModalDialog` returns with no further action.

The `CustomGetFile` and `CustomPutFile` procedures call your filter function to process events in both the main dialog box and any subsidiary dialog boxes (such as the dialog box for naming a new folder while saving a document through `CustomPutFile`). To determine whether the dialog record describes the main dialog box or a subsidiary dialog box, check the value of the `refCon` field in the window record in the dialog record, as described in "Writing a Dialog Hook Function" beginning on page 3-21.

Listing 3-10 shows how to define a modal-dialog filter function that prevents update events from clogging the event queue.

Listing 3-10 A sample modal-dialog filter function

```
FUNCTION MyModalFilter (theDialog: DialogPtr; VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
BEGIN
   MyModalFilter := FALSE;             {we haven't handled the event yet}
   IF theEvent.what = updateEvt THEN
      IF IsAppWindow(WindowPtr(theEvent.message)) THEN
         BEGIN
            DoUpdateEvent(WindowPtr(theEvent.message));
            MyModalFilter := TRUE;     {we have handled the event}
         END;
END;
```

If this filter function receives an update event for a window other than the Standard File Package dialog box, it calls the application's routine for handling update events (`DoUpdateEvent`) and returns `TRUE` to indicate that the event has been handled. See the chapters "Event Manager" and "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for complete details on handling update events.

Writing an Activation Procedure

The **activation procedure** controls the highlighting of dialog items that are defined by your application and can receive keyboard input. Ordinarily, you need to supply an activation procedure only if your application builds a list from which the user can select entries. The Standard File Package supplies the activation procedure for the file display list and for all TextEdit fields. You can also use the activation procedure to keep track of which field is receiving keyboard input, if your application needs that information.

The target of keyboard input is called the **active field.** The two standard keyboard-input fields are the filename field (present only in Save dialog boxes) and the display list. Unless you override it through your own dialog hook function, the Standard File Package handles the highlighting of its own items and TextEdit fields. When the user changes the keyboard target by pressing the mouse button or the Tab key, the Standard File Package calls your activation procedure twice: the first call specifies which field is being

deactivated, and the second specifies which field is being activated. Your application is responsible for removing the highlighting when one of its fields becomes inactive and for adding the highlighting when one of its fields becomes active. The Standard File Package can handle the highlighting of all TextEdit fields, even those defined by your application.

The activation procedure receives four parameters: a dialog pointer, a dialog item number, a Boolean value that specifies whether the field is being activated (TRUE) or deactivated (FALSE), and a pointer to your own data.

```
PROCEDURE MyActivateProc (theDialog: DialogPtr; itemNo: Integer;
                          activating: Boolean; myDataPtr: Ptr);
```

Setting the Current Directory

The first time your application calls one of the Standard File Package routines, the default current directory (that is, the directory whose contents are listed in the dialog box) is determined by the way in which your application was launched.

- If the user launched your application directly (perhaps by double-clicking its icon in the Finder), the default directory is the directory in which your application is located.
- If the user launched your application indirectly (perhaps by double-clicking one of your application's document icons), the default directory is the directory in which that document is located.

At each subsequent call to one of the Standard File Package routines, the default current directory is simply the directory that was current when the user completed the previous dialog box. You can use the function `GetSFCurDir` defined in Listing 3-11 to determine the current directory.

Listing 3-11 Determining the current directory

```
FUNCTION GetSFCurDir: LongInt;
TYPE
   LongIntPtr = ^LongInt;
CONST
   CurDirStore = $398;
BEGIN
   GetSFCurDir := LongIntPtr(CurDirStore)^;
END;
```

You can use the `GetSFCurVol` function defined in Listing 3-12 to determine the current volume.

Listing 3-12 Determining the current volume

```
FUNCTION GetSFCurVol: Integer;
TYPE
   IntPtr = ^Integer;
CONST
   SFSaveDisk = $214;
BEGIN
   GetSFCurVol := -IntPtr(SFSaveDisk)^;
END;
```

If necessary, you can change the default current directory and volume. For example, when the user needs to select a dictionary file for a spell-checking application, the application might set the current directory to a directory containing document-specific dictionary files. This saves the user from having to navigate the directory hierarchy from the directory containing documents to that containing dictionary files. You can use the procedure `SetSFCurDir` defined in Listing 3-13 to set the current directory.

Listing 3-13 Setting the current directory

```
PROCEDURE SetSFCurDir (dirID: LongInt);
TYPE
   LongIntPtr = ^LongInt;
CONST
   CurDirStore = $398;
BEGIN
   LongIntPtr(CurDirStore)^ := dirID;
END;
```

You can use the procedure `SetSFCurVol` defined in Listing 3-14 to set the current volume.

Listing 3-14 Setting the current volume

```
PROCEDURE SetSFCurVol (vRefNum: Integer);
TYPE
   IntPtr = ^Integer;
CONST
   SFSaveDisk = $214;
BEGIN
   IntPtr(SFSaveDisk)^ := -vRefNum;
END;
```

Note
Most applications don't need to alter the default current directory or volume. ◆

If you are using the enhanced Standard File Package routines, you can set the current directory by filling in the fields of the file system specification in the reply record passed to `CustomGetFile` or `CustomPutFile`. You do this within your dialog hook function. Listing 3-15 defines a dialog hook function that makes the currently active System Folder the current directory.

Listing 3-15 Setting the current directory

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
                    Integer;
VAR
   myReplyPtr:    StandardFileReplyPtr;
   foundVRefNum:  Integer;
   foundDirID:    LongInt;
   myErr:         OSErr;
BEGIN
   MyDlgHook := item;            {by default, return the item passed in}
   IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
      Exit(MyDlgHook);           {this function is only for main dialog box}

   CASE item OF
      sfHookFirstCall:           {pseudo-item: first time function called}
         BEGIN
            myReplyPtr := StandardFileReplyPtr(myDataPtr);
            myErr := FindFolder(kOnSystemDisk, kSystemFolderType,
                           kDontCreateFolder, foundVRefNum, foundDirID);
            IF myErr = noErr THEN
               BEGIN
                  myReplyPtr^.sfFile.parID := foundDirID;
                  myReplyPtr^.sfFile.vRefNum := foundVRefNum;
                  MyDlgHook := sfHookChangeSelection;
               END;
         END;
      OTHERWISE
         ;                       {ignore all other items}
   END;
END;
```

This dialog hook function installs the System Folder's volume reference number and parent directory ID into the file system specification whose address is passed in the `myDataPtr` parameter. Because the dialog hook function returns the constant `sfHookChangeSelection` the first time it is called (that is, in response to the `sfHookFirstCall` pseudo-item), the Standard File Package sets the current directory to the indicated directory when the dialog box is displayed.

Selecting a Directory

You can present the recommended user interface for selecting a directory by calling the `CustomGetFile` procedure and passing it the addresses of a custom file filter function and a dialog hook function. See "Selecting Volumes and Directories" on page 3-10 for a description of the appearance and behavior of the directory selection dialog box.

The file filter function used to select directories is quite simple; it ensures that only directories, not files, are listed in the dialog box displayed by `CustomGetFile`. Listing 3-16 defines a file filter function you can use for this purpose.

Listing 3-16 A file filter function that lists only directories

```
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr): Boolean;
CONST
   kFolderBit = 4;                  {bit set in ioFlAttrib for a directory}
BEGIN                               {list directories only}
   MyCustomFileFilter := NOT BTst(pb^.ioFlAttrib, kFolderBit);
END;
```

The function `MyCustomFileFilter` simply inspects the appropriate bit in the file attributes (`ioFlAttrib`) field of the catalog information parameter block passed to it. If the directory bit is set, the file filter function returns `FALSE`, indicating that the item should appear in the list; otherwise, the file filter function returns `TRUE` to exclude the item from the list. Because a volume is identified via its root directory, volumes also appear in the list of items in the dialog box.

The title of the Select button should identify which directory is available for selection. You can use the `SetButtonTitle` procedure defined in Listing 3-17 to set the title of a button.

Your dialog hook function calls the `SetButtonTitle` procedure to copy the truncated title of the selected item into the Select button. This title eliminates possible user confusion about which directory is available for selection. If no item in the list is selected, the dialog hook function uses the name of the directory shown in the pop-up menu as the title of the Select button.

Listing 3-17 Setting a button's title

```
PROCEDURE SetButtonTitle (ButtonHdl: Handle; name: Str255; ButtonRect: Rect);
VAR
   result:  Integer;    {result of TruncString}
   width:   Integer;    {width available for name of directory}
BEGIN
   gPrevSelectedName := name;
   WITH ButtonRect DO
      width := (right - left) - (StringWidth('Select ""') + CharWidth(' '));
   result := TruncString(width, name, smTruncMiddle);
   SetCTitle(ControlHandle(ButtonHdl), CONCAT('Select "', name, '"'));
   ValidRect(ButtonRect);
END;
```

The `SetButtonTitle` procedure is passed a handle to the button whose title is to be changed, the name of the directory available for selection, and the button's enclosing rectangle. The global variable `gPrevSelectedName` holds the full directory name, before truncation.

A dialog hook function manages most of the process of letting the user select a director. Listing 3-18 defines a dialog hook function that handles user selections in the dialog box.

Listing 3-18 Handling user selections in the directory selection dialog box

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
                   Integer;
CONST
   kGetDirBTN     = 10;          {Select directory button}
TYPE
   SFRPtr         = ^StandardFileReply;
VAR
   myType:        Integer;       {menu item selected}
   myHandle:      Handle;        {needed for GetDItem}
   myRect:        Rect;          {needed for GetDItem}
   myName:        Str255;
   myPB:          CInfoPBRec;
   mySFRPtr:      SFRPtr;
   myErr:         OSErr;
BEGIN
   MyDlgHook := item;      {default, except in special cases below}
   IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
      Exit(MyDlgHook);     {this function is only for main dialog box}

   GetDItem(theDialog, kGetDirBTN, myType, myHandle, myRect);
   IF item = sfHookFirstCall THEN
```

```
      BEGIN
         {Determine current folder name and set title of Select button.}
         WITH myPB DO
            BEGIN
               ioCompletion := NIL;
               ioNamePtr := @myName;
               ioVRefNum := GetSFCurVol;
               ioFDirIndex := - 1;
               ioDirID := GetSFCurDir;
            END;
         myErr := PBGetCatInfo(@myPB, FALSE);
         SetButtonTitle(myHandle, myName, myRect);
      END
   ELSE
      BEGIN
         {Get mySFRPtr from 3rd parameter to hook function.}
         mySFRPtr := SFRPtr(myDataPtr);
         {Track name of folder that can be selected.}
         IF (mySFRPtr^.sfIsFolder) OR (mySFRPtr^.sfIsVolume) THEN
            myName := mySFRPtr^.sfFile.name
         ELSE
            BEGIN
               WITH myPB DO
                  BEGIN
                     ioCompletion := NIL;
                     ioNamePtr := @myName;
                     ioVRefNum := mySFRPtr^.sfFile.vRefNum;
                     ioFDirIndex := -1;
                     ioDrDirID := mySFRPtr^.sfFile.parID;
                  END;
               myErr := PBGetCatInfo(@myPB, FALSE);
            END;
         {Change directory name in button title as needed.}
         IF myName <> gPrevSelectedName THEN
            SetButtonTitle(myHandle, myName, myRect);

         CASE item OF
            kGetDirBTN:                    {force return by faking a cancel}
               MyDlgHook := sfItemCancelButton;
            sfItemCancelButton:
               gDirSelectionFlag := FALSE;{flag no directory was selected}
            OTHERWISE
               ;
         END; {CASE}
      END;
END;
```

The MyDlgHook dialog hook function defined in Listing 3-18 calls the File Manager function PBGetCatInfo to retrieve the name of the directory to be selected. When the dialog hook function is first called (that is, when item is set to sfHookFirstCall), MyDlgHook determines the current volume and directory by calling the functions GetSFCurVol and GetSFCurDir. When MyDlgHook is called each subsequent time, MyDlgHook calls PBGetCatInfo with the volume reference number and directory ID of the previously opened directory.

When the user clicks the Select button, MyDlgHook returns the item sfItemCancelButton. When the user clicks the real Cancel button, MyDlgHook sets the global variable gDirSelectionFlag to FALSE, indicating that the user didn't select a directory. The function DoGetDirectory uses that variable to distinguish between clicks of Cancel and clicks of Select.

The function DoGetDirectory defined in Listing 3-19 uses the file filter function and the dialog hook functions defined above to manage the directory selection dialog box. On exit, DoGetDirectory returns a standard file reply record describing the selected directory.

Listing 3-19 Presenting the directory selection dialog box

```
FUNCTION DoGetDirectory: StandardFileReply;
VAR
   myReply:          StandardFileReply;
   myTypes:          SFTypeList;          {types of files to display}
   myPoint:          Point;               {upper-left corner of box}
   myNumTypes:       Integer;
   myModalFilter:    ModalFilterYDProcPtr;
   myActiveList:     Ptr;
   myActivateProc:   ActivateYDProcPtr;
   myName:           Str255;
CONST
   rGetDirectoryDLOG = 128;               {resource ID of custom dialog box}
BEGIN
   gPrevSelectedName := '';         {initialize name of previous selection}
   gDirSelectionFlag := TRUE;       {initialize directory selection flag}
   myNumTypes := -1;                {pass all types of files to file filter}
   myPoint.h := -1;                 {center dialog box on screen}
   myPoint.v := -1;
   myModalFilter := NIL;
   myActiveList := NIL;
   myActivateProc := NIL;

   CustomGetFile(@MyCustomFileFilter, myNumTypes, myTypes, myReply,
                 rGetDirectoryDLOG, myPoint, @MyDlgHook, myModalFilter,
                 myActiveList, myActivateProc, @myReply);
```

```
   {Get the name of the directory.}
   IF gDirSelectionFlag AND myReply.sfIsVolume THEN
      myName := Concat(myReply.sfFile.name, ':')
   ELSE
      myName := myReply.sfFile.name;

   IF gDirSelectionFlag AND myReply.sfIsVolume THEN
      myReply.sfFile.name := myName
   ELSE IF gDirSelectionFlag THEN
      myReply.sfFile.name := gPrevSelectedName;
   gDirSelectionFlag := FALSE;
   DoGetDirectory := myReply;
END;
```

The `DoGetDirectory` function initializes the two global variables `gPrevSelectedName` and `gDirSelectionFlag`. As you have seen, these two variables are used by the custom dialog hook function. Then `DoGetDirectory` calls `CustomGetFile` to display the directory selection dialog box and handle user selections. When the user selects a directory or clicks the Cancel button, the dialog hook function returns `sfItemCancelButton` and `CustomGetFile` exits. At that point, the reply record contains information about the last item selected in the list of available items.

Selecting a Volume

You can present the recommended user interface for selecting a volume by calling the `CustomGetFile` procedure and passing it the addresses of a custom file filter function and a dialog hook function. See “Selecting Volumes and Directories” on page 3-10 for a description of the appearance and behavior of the volume selection dialog box.

The file filter function used to select volumes is quite simple; it ensures that only volumes, not files or directories, are listed in the dialog box displayed by `CustomGetFile`. Listing 3-16 defines a file filter function you can use to do this.

Listing 3-20 A file filter function that lists only volumes

```
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr): Boolean;
CONST
   kFolderBit = 4;                  {bit set in ioFlAttrib for a directory}
BEGIN                               {list volumes only}
   MyCustomFileFilter := TRUE;      {assume you don't list the item}
   IF BTst(pb^.ioFlAttrib, kFolderBit) AND (pb^.ioDrParID = fsRtParID) THEN
      MyCustomFileFilter := FALSE;
END;
```

The function `MyCustomFileFilter` inspects the appropriate bit in the file attributes (`ioFlAttrib`) field of the catalog information parameter block passed to it. If the directory bit is set, `MyCustomFileFilter` checks whether the parent directory ID of the directory is equal to `fsRtParID`, which is always the parent directory ID of a volume's root directory. If it is, the file filter function returns `FALSE`, indicating that the item should appear in the list of volumes; otherwise, the file filter function returns `TRUE` to exclude the item from the list.

A dialog hook function for handling the items in the volume selection dialog box is defined in Listing 3-21.

Listing 3-21 Handling user selections in the volume selection dialog box

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
                    Integer;
VAR
   myType:         Integer;       {menu item selected}
   myHandle:       Handle;        {needed for GetDItem}
   myRect:         Rect;          {needed for GetDItem}
   myName:         Str255;        {new title for Open button}
BEGIN
   MyDlgHook := item;      {default, except in special cases below}
   IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
      Exit(MyDlgHook);     {this function is only for main dialog box}

   CASE item OF
      sfHookFirstCall:
         BEGIN
            {Set button title and go to desktop.}
            myName := 'Select';
            GetDItem(theDialog, sfItemOpenButton, myType, myHandle, myRect);
            SetCTitle(ControlHandle(myHandle), myName);
            MyDlgHook := sfHookGoToDesktop;
         END;
      sfHookGoToDesktop:         {map Cmd-D to a null event}
         MyDlgHook := sfHookNullEvent;
      sfHookChangeSelection:
         MyDlgHook := sfHookGoToDesktop;
      sfHookGoToNextDrive:       {map Cmd-Left Arrow to a null event}
         MyDlgHook := sfHookNullEvent;
      sfHookGoToPrevDrive:       {map Cmd-Right Arrow to a null event}
         MyDlgHook := sfHookNullEvent;
      sfItemOpenButton, sfHookOpenFolder:
         MyDlgHook := sfItemOpenButton;
      OTHERWISE
         ;
   END;
END;
```

You can prompt the user to select a volume by calling the function `DoGetVolume` defined in Listing 3-22.

Listing 3-22 Presenting the volume selection dialog box

```
FUNCTION DoGetVolume: StandardFileReply;
VAR
   myReply:          StandardFileReply;
   myTypes:          SFTypeList;          {types of files to display}
   myPoint:          Point;               {upper-left corner of box}
   myNumTypes:       Integer;
   myModalFilter:    ModalFilterYDProcPtr;
   myActiveList:     Ptr;
   myActivateProc:   ActivateYDProcPtr;
CONST
   rGetVolumeDLOG    = 129;               {resource ID of custom dialog box}
BEGIN
   myNumTypes := -1;                      {pass all types of files}
   myPoint.h := -1;                       {center dialog box on screen}
   myPoint.v := -1;
   myModalFilter := NIL;
   myActiveList := NIL;
   myActivateProc := NIL;

   CustomGetFile(@MyCustomFileFilter, myNumTypes, myTypes, myReply,
                 rGetVolumeDLOG, myPoint, @MyDlgHook, myModalFilter,
                 myActiveList, myActivateProc, @myReply);

   DoGetVolume := myReply;
END;
```

Using the Original Procedures

The Standard File Package still recognizes all procedures available before system software version 7.0 (`SFGetFile`, `SFPutFile`, `SFPGetFile`, and `SFPPutFile`). It displays the new interface for all applications that don't customize the dialog boxes in incompatible ways (that is, applications that specify both the dialog hook and the modal-dialog filter pointers as `NIL` and that specify no alternative dialog ID).

When the Standard File Package can't use the enhanced dialog box layout because an application customized the dialog box with the earlier procedures, it nevertheless makes some changes to the display:

- It changes the label of the Drive button to Desktop and makes the desktop the root of the display.
- It moves the volume icon slightly to the right, to make room for selection highlighting around the display list field.

If, however, a customized dialog box has suppressed the file display list (by specifying coordinates outside of the dialog box), the Standard File Package uses the earlier interface, on the assumption that the dialog box is designed for volume selection.

If you need to use the procedures available before system software version 7.0, you need to be aware of a number of differences between those procedures and the enhanced procedures. These are the most important differences:

- The original procedures do not recognize some pseudo-items under previous system software versions. For example, the pseudo-item `sfHookLastCall` is not used before version 7.0. See the comments under "Constants" in "Summary of the Standard File Package" (beginning on page 3-60) for information on which pseudo-items are universally available.
- The original standard file reply record (type `SFReply`) returns a working directory reference number, not a volume reference number. Typically, you should immediately convert that number to a volume reference number and directory ID using `GetWDInfo` or `PBGetWDInfo`. Then close the working directory by calling `CloseWD` or `PBCloseWD`. For details on these functions, see the chapter "File Manager" in this book.
- Dialog hook functions used with the original procedures are not passed a `myDataPtr` parameter.

Standard File Package Reference

This section describes the data structures and routines that are specific to the Standard File Package. The "Data Structures" section shows the Pascal data structures for the original and the enhanced Standard File reply records. The section "Standard File Package Routines" describes routines for opening and saving files. The section "Application-Defined Routines" describes the routines that your application can define to customize the operations of the Standard File Package routines.

Data Structures

The Standard File Package exchanges information with your application using a standard file reply record. If you use the procedures introduced in system software version 7.0, you use a reply record of type `StandardFileReply`. If you use the procedures available before version 7.0, you must use a reply record of type `SFReply`.

Enhanced Standard File Reply Record

When you use one of the procedures `StandardPutFile`, `StandardGetFile`, `CustomPutFile`, or `CustomGetFile`, you pass a reply record of type `StandardFileReply`.

```
TYPE StandardFileReply =
RECORD
   sfGood:        Boolean;    {TRUE if user did not cancel}
   sfReplacing:   Boolean;    {TRUE if replacing file with same name}
   sfType:        OSType;     {file type}
   sfFile:        FSSpec;     {selected file, folder, or volume}
   sfScript:      ScriptCode; {script of file, folder, or volume name}
   sfFlags:       Integer;    {Finder flags of selected item}
   sfIsFolder:    Boolean;    {selected item is a folder}
   sfIsVolume:    Boolean;    {selected item is a volume}
   sfReserved1:   LongInt;    {reserved}
   sfReserved2:   Integer;    {reserved}
END;
```

Field descriptions

`sfGood` Reports whether the reply record is valid. The value is `TRUE` after the user clicks Save or Open; `FALSE` after the user clicks Cancel. When the user has completed the dialog box, the other fields in the reply record are valid only if the `sfGood` field contains `TRUE`.

`sfReplacing` Reports whether a file to be saved replaces an existing file of the same name. This field is valid only after a call to the `StandardPutFile` or `CustomPutFile` procedure. When the user assigns a name that duplicates that of an existing file, the Standard File Package asks for verification by displaying a subsidiary dialog box (illustrated in Figure 3-4, page 3-7). If the user verifies the name, the Standard File Package sets the `sfReplacing` field to `TRUE` and returns to your application; if the user cancels the overwriting of the file, the Standard File Package returns to the main dialog box. If the name does not conflict with an existing name, the Standard File Package sets the field to `FALSE` and returns.

`sfType` Contains the file type of the selected file. (File types are described in the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.) Only `StandardGetFile` and `CustomGetFile` return a file type in this field.

`sfFile` Describes the selected file, folder, or volume with a file system specification record, which contains a volume reference number, parent directory ID, and name. (See the chapter "File Manager" in this book for a complete description of the file system specification record.) If the selected item is an alias for another item, the Standard

File Package resolves the alias and places the file system specification record for the target in the `sfFile` field when the user completes the dialog box. If the selected file is a stationery pad, the reply record describes the file itself, not a copy of the file.

`sfScript` Identifies the script in which the name of the document is to be displayed. (This information is used by the Finder and by the Standard File Package.) A script code of `smSystemScript` (`–1`) represents the default system script.

`sfFlags` Contains the Finder flags from the Finder information record in the catalog entry for the selected file. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a description of the Finder flags.) This field is returned only by `StandardGetFile` and `CustomGetFile`. If your application supports stationery, it should check the stationery bit in the Finder flags to determine whether to treat the selected file as stationery. Unlike the Finder, the Standard File Package does not automatically create a document from a stationery pad and pass your application the new document. If the user opens a stationery document from within an application that does not support stationery, the Standard File Package displays a dialog box warning the user that the master copy is being opened.

`sfIsFolder` Reports whether the selected item is a folder (`TRUE`) or a file or volume (`FALSE`). This field is meaningful only during the execution of a dialog hook function.

`sfIsVolume` Reports whether the selected item is a volume (`TRUE`) or a file or folder (`FALSE`). This field is meaningful only during the execution of a dialog hook function.

`sfReserved1` Reserved.

`sfReserved2` Reserved.

Original Standard File Reply Record

When you use one of the original Standard File Package procedures `SFPutFile`, `SFGetFile`, `SFPPutFile`, or `SFPGetFile`, you pass a reply record of type `SFReply`.

```
SFReply =
RECORD
   good:       Boolean;    {TRUE if user did not cancel}
   copy:       Boolean;    {reserved}
   fType:      OSType;     {file type}
   vRefNum:    Integer;    {working directory reference number}
   version:    Integer;    {reserved}
   fName:      Str63;      {filename}
END;
```

Field descriptions

`good`	Reports whether the reply record is valid. The value is `TRUE` after the user clicks Save or Open; `FALSE` after the user clicks Cancel. When the user has completed the dialog box, the other fields in the reply record are valid only if the value of `good` is `TRUE`.
`copy`	Reserved.
`fType`	Contains the file type of the selected file. (File types are described in the chapter "Finder Interface" of *Inside Macintosh: Macintosh Toolbox Essentials*.) Only `SFGetFile` and `SFPGetFile` return a file type in this field.
`vRefNum`	Contains the working directory reference number of the selected file.
`version`	Reserved.
`fName`	Contains the name of the selected file.

Note

In spite of its name, the `vRefNum` field does not contain a volume reference number. Instead, it contains a working directory reference number, which encodes both the volume reference number and the parent directory ID of the selected file. You can obtain the volume reference number and directory ID of the file by calling `GetWDInfo` or `PBGetWDInfo`. See the chapter "File Manager" in this book for details about working directory reference numbers. ◆

Standard File Package Routines

This section describes the routines you can use to prompt the user for a file's name and location after a request to save or open a file. If your application is designed to run in system software versions prior to version 7.0, you must use either `SFGetFile` or `SFPGetFile` when opening a file and either `SFPutFile` or `SFPPutFile` when saving a file.

If your application is designed to take advantage of features introduced in system software version 7.0 or later, you can use the new routines intended to simplify the code required to elicit a filename from the user. The `StandardPutFile` and `StandardGetFile` procedures are simplified versions of the original procedures for handling the user interface during the storing and retrieving of files. The `CustomPutFile` and `CustomGetFile` procedures are customizable versions of the same procedures.

Saving Files

You can use the `StandardPutFile` procedure to present the standard user interface when the user asks to save a file. If you need to add elements to the default dialog boxes or exercise greater control over user actions in the dialog box, use `CustomPutFile`.

If your application is designed to execute in system software versions earlier than version 7.0, you can use the corresponding procedures `SFPutFile` and `SFPPutFile`.

StandardPutFile

You can use the `StandardPutFile` procedure to display the default Save dialog box when the user is saving a file.

```
PROCEDURE StandardPutFile (prompt: Str255; defaultName: Str255;
                           VAR reply: StandardFileReply);
```

`prompt` The prompt message to be displayed over the text field.

`defaultName` The initial name of the file.

`reply` The reply record, which `StandardPutFile` fills in before returning.

DESCRIPTION

The `StandardPutFile` procedure presents a dialog box through which the user specifies the name and location of a file to be written to. The dialog box is centered on the screen. While the dialog box is active, `StandardPutFile` gets and handles events until the user completes the interaction, either by selecting a name and authorizing the save or by canceling the save. The `StandardPutFile` procedure returns the user's input in a record of type `StandardFileReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `StandardPutFile` are

Trap macro	Selector
`_Pack3`	$0005

SPECIAL CONSIDERATIONS

The `StandardPutFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `StandardPutFile` is available before calling it.

Because `StandardPutFile` may move memory, you should not call it at interrupt time.

CustomPutFile

Use the `CustomPutFile` procedure when your application requires more control over the Save dialog box than is possible using `StandardPutFile`.

```
PROCEDURE CustomPutFile (prompt: Str255; defaultName: Str255;
                         VAR reply: StandardFileReply;
                         dlgID: Integer; where: Point;
                         dlgHook: DlgHookYDProcPtr;
                         filterProc: ModalFilterYDProcPtr;
                         activeList: Ptr;
                         activateProc: ActivateYDProcPtr;
                         yourDataPtr: UNIV Ptr);
```

`prompt` The prompt message to be displayed over the text field.

`defaultName` The initial name of the file.

`reply` The reply record, which `CustomPutFile` fills in before returning.

`dlgID` The resource ID of a customized dialog template. To use the standard template, set this parameter to 0.

`where` The upper-left corner of the dialog box, in global coordinates. If you specify the point (–1,–1), `CustomPutFile` automatically centers the dialog box on the screen.

`dlgHook` A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you have not added any items to the dialog box and want the standard items handled in the standard ways. See "Writing a Dialog Hook Function" on page 3-21 for a description of the dialog hook function.

`filterProc` A pointer to your modal-dialog filter function, which determines how the `ModalDialog` procedure filters events when called by the `CustomPutFile` procedure. Specify a value of `NIL` if you are not supplying your own function. See "Writing a Modal-Dialog Filter Function" on page 3-28 for a description of the modal-dialog filter function.

`activeList` A pointer to a list of all dialog items that can be activated—that is, can be the target of keyboard input. If you supply an `activeList` parameter of `NIL`, `CustomPutFile` uses the default targets (the filename field and the list of files and folders). If you have added any fields that can accept keyboard input, you must modify the list. The list is stored as an array of 16-bit integers. The first integer is the number of items in the list. The remaining integers are the item numbers of all possible keyboard targets, in the order that they are activated by the Tab key.

`activateProc`

A pointer to your activation procedure, which controls the highlighting of dialog items that are defined by your application and that can receive keyboard input. See "Writing an Activation Procedure" on page 3-30 for a description of the activation procedure.

`yourDataPtr`

Any 4-byte value; usually, a pointer to optional data supplied by your application. When `CustomPutFile` calls any of your callback routines, it adds this parameter, making the data available to your callback routines. If you are not supplying any data of your own, you can specify a value of `NIL`.

DESCRIPTION

The `CustomPutFile` procedure is an alternative to `StandardPutFile` when you want to display a customized Save dialog box or handle the default dialog box in a customized way. During the dialog, `CustomPutFile` gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a name and authorizing the save operation or by canceling the save operation. The `CustomPutFile` procedure returns the user's input in a record of type `StandardFileReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `CustomPutFile` are

Trap macro	Selector
`_Pack3`	$0007

SPECIAL CONSIDERATIONS

The `CustomPutFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `CustomPutFile` is available before calling it.

Because `CustomPutFile` may move memory, you should not call it at interrupt time.

SFPutFile

Use the `SFPutFile` procedure to display the standard Save dialog box when the user is saving a file.

```
PROCEDURE SFPutFile (where: Point; prompt: Str255;
                     origName: Str255; dlgHook: DlgHookProcPtr;
                     VAR reply: SFReply);
```

`where`	The upper-left corner of the dialog box, in global coordinates.
`prompt`	The prompt message to be displayed over the text field.
`origName`	The initial name of the file.
`dlgHook`	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you want the standard items handled in the standard ways. See "Writing a Dialog Hook Function" on page 3-21 for a description of the dialog hook function.
`reply`	The reply record, which `SFPutFile` fills in before returning.

DESCRIPTION

The `SFPutFile` procedure presents a dialog box through which the user specifies the name and location of a file to be written to. During the dialog, `SFPutFile` gets and handles events until the user completes the interaction, either by selecting a name and authorizing the save or by canceling the save. The `SFPutFile` procedure returns the user's input in a record of type `SFReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SFPutFile` are

Trap macro	Selector
`_Pack3`	$0001

SPECIAL CONSIDERATIONS

Because `SFPutFile` may move memory, you should not call it at interrupt time.

SFPPutFile

Use the `SFPPutFile` procedure when your application requires more control over the Save dialog box than is possible using `SFPutFile`.

```
PROCEDURE SFPPutFile (where: Point; prompt: Str255;
                      origName: Str255; dlgHook: DlgHookProcPtr;
                      VAR reply: SFReply; dlgID: Integer;
                      filterProc: ModalFilterProcPtr);
```

`where`	The upper-left corner of the dialog box, in global coordinates.
`prompt`	The prompt message to be displayed over the text field.
`origName`	The initial name of the file, if any.

`dlgHook` A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you have not added any items to the dialog box and want the standard items handled in the standard ways. See "Writing a Dialog Hook Function" on page 3-21 for a description of the dialog hook function.

`reply` The reply record, which `SFPPutFile` fills in before returning.

`dlgID` The resource ID of a customized dialog template. To use the standard template, set this parameter to –3999.

`filterProc` A pointer to your modal-dialog filter function, which determines how the `ModalDialog` procedure filters events when called by the `SFPPutFile` procedure. Specify a value of `NIL` if you are not supplying your own function. See "Writing a Modal-Dialog Filter Function" on page 3-28 for a description of the modal-dialog filter function.

DESCRIPTION

The `SFPPutFile` procedure is an alternative to `SFPutFile` when you want to display a customized Save dialog box or handle the default dialog box in a customized way. During the dialog, `SFPPutFile` gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a name and authorizing the save operation or by canceling the save operation. `SFPPutFile` returns the user's input in a record of type `SFReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SFPPutFile` are

Trap macro	Selector
`_Pack3`	$0003

SPECIAL CONSIDERATIONS

Because `SFPPutFile` may move memory, you should not call it at interrupt time.

Opening Files

You can use the `StandardGetFile` procedure to present the standard user interface when the user asks to open a file. If you need to add elements to the default dialog boxes or exercise greater control over user actions in the dialog box, use `CustomGetFile`.

If your application is designed to execute in system software versions earlier than version 7.0, you can use the corresponding procedures `SFGetFile` and `SFPGetFile`.

StandardGetFile

You can use the `StandardGetFile` procedure to display the default Open dialog box when the user is opening a file.

```
PROCEDURE StandardGetFile (fileFilter: FileFilterProcPtr;
                           numTypes: Integer;
                           typeList: SFTypeList;
                           VAR reply: StandardFileReply);
```

`fileFilter` A pointer to an optional file filter function, provided by your application, through which `StandardGetFile` passes files of the specified types.

`numTypes` The number of file types to be displayed. If you specify a `numTypes` value of –1, the first filtering passes files of all types.

`typeList` A list of file types to be displayed.

`reply` The reply record, which `StandardGetFile` fills in before returning.

DESCRIPTION

The `StandardGetFile` procedure presents a dialog box through which the user specifies the name and location of a file to be opened. While the dialog box is active, `StandardGetFile` gets and handles events until the user completes the interaction, either by selecting a file to open or by canceling the operation. `StandardGetFile` returns the user's input in a record of type `StandardFileReply`.

The `fileFilter`, `numTypes`, and `typeList` parameters together determine which files appear in the displayed list. The first filtering is by file type, which you specify in the `numTypes` and `typeList` parameters. The `numTypes` parameter specifies the number of file types to be displayed. You can specify one or more types. If you specify a `numTypes` value of –1, the first filtering passes files of all types.

The `fileFilter` parameter points to an optional file filter function, provided by your application, through which `StandardGetFile` passes files of the specified types. See "Writing a File Filter Function" on page 3-20 for a description of the file filter function.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `StandardGetFile` are

Trap macro	Selector
`_Pack3`	$0006

SPECIAL CONSIDERATIONS

The `StandardGetFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `StandardGetFile` is available before calling it.

Because `StandardGetFile` may move memory, you should not call it at interrupt time.

CustomGetFile

Call the `CustomGetFile` procedure when your application requires more control over the Open dialog box than is possible using `StandardGetFile`.

```
PROCEDURE CustomGetFile (fileFilter: FileFilterYDProcPtr;
                         numTypes: Integer;
                         typeList: SFTypeList;
                         VAR reply: StandardFileReply;
                         dlgID: Integer;
                         where: Point;
                         dlgHook: DlgHookYDProcPtr;
                         filterProc: ModalFilterYDProcPtr;
                         activeList: Ptr;
                         activateProc: ActivateYDProcPtr;
                         yourDataPtr: UNIV Ptr);
```

`fileFilter` A pointer to an optional file filter function, provided by your application, through which `CustomGetFile` passes files of the specified types.

`numTypes` The number of file types to be displayed. If you specify a `numTypes` value of –1, the first filtering passes files of all types.

`typeList` A list of file types to be displayed.

`reply` The reply record, which `CustomGetFile` fills in before returning.

`dlgID` The resource ID of a customized dialog template. To use the standard template, set this parameter to 0.

`where` The upper-left corner of the dialog box in global coordinates. If you specify the point (–1,–1), `CustomGetFile` automatically centers the dialog box on the screen.

`dlgHook` A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you have not added any items to the dialog box and want the standard items handled in the standard ways. See “Writing a Dialog Hook Function” on page 3-21 for a description of the dialog hook function.

`filterProc` A pointer to your modal-dialog filter function, which determines how `ModalDialog` filters events when called by `CustomGetFile`. Specify a value of `NIL` if you are not supplying your own function. See “Writing a Modal-Dialog Filter Function” on page 3-28 for a description of the modal-dialog filter function.

`activeList` A pointer to a list of all dialog items that can be activated—that is, made the target of keyboard input. The list is stored as an array of 16-bit integers. The first integer is the number of items in the list. The remaining integers are the item numbers of all possible keyboard targets, in the order that they are activated by the Tab key. If you supply an `activeList` parameter of `NIL`, `CustomGetFile` directs all keyboard input to the displayed list.

`activateProc`

A pointer to your activation procedure, which controls the highlighting of dialog items that are defined by your application and that can receive keyboard input. See "Writing an Activation Procedure" on page 3-30 for a description of the activation procedure.

`yourDataPtr`

A pointer to optional data supplied by your application. When `CustomGetFile` calls any of your callback routines, it pushes this parameter on the stack, making the data available to your callback routines. If you are not supplying any data of your own, specify a value of `NIL`.

DESCRIPTION

The `CustomGetFile` procedure is an alternative to `StandardGetFile` when you want to use a customized dialog box or handle the default Open dialog box in a customized way. `CustomGetFile` presents a dialog box through which the user specifies the name and location of a file to be opened. While the dialog box is active, `CustomGetFile` gets and handles events until the user completes the interaction, either by selecting a file to open or by canceling the operation. `CustomGetFile` returns the user's input in a record of type `StandardFileReply`.

The first four parameters are similar to the same parameters in `StandardGetFile`. The `fileFilter`, `numTypes`, and `typeList` parameters determine which files appear in the list of choices. If you specify a value of –1 in the `numTypes` parameter, `CustomGetFile` displays or passes to your file filter function all files and folders (not just the files) at the current level of the display hierarchy. If you provide a filter function, `CustomGetFile` passes it both the pointer to the catalog entry for each file to be processed and also a pointer to the optional data passed by your application in its call to `CustomGetFile`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `CustomGetFile` are

Trap macro	Selector
`_Pack3`	$0008

SPECIAL CONSIDERATIONS

The `CustomGetFile` procedure is not available in all versions of system software. Use the `Gestalt` function to determine whether `CustomGetFile` is available before calling it.

Because `CustomGetFile` may move memory, you should not call it at interrupt time.

SFGetFile

Use the SFGetFile procedure to display the default Open dialog box when the user is opening a file.

```
PROCEDURE SFGetFile (where: Point; prompt: Str255;
                     fileFilter: FileFilterProcPtr;
                     numTypes: Integer; typeList: SFTypeList;
                     dlgHook: DlgHookProcPtr; VAR reply: SFReply);
```

where	The upper-left corner of the dialog box, in global coordinates.
prompt	Ignored.
fileFilter	A pointer to an optional file filter function, provided by your application, through which SFGetFile passes files of the specified types.
numTypes	The number of file types to be displayed. If you specify a numTypes value of –1, the first filtering passes files of all types.
typeList	A list of file types to be displayed.
dlgHook	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of NIL if you want the standard items handled in the standard ways.
reply	The reply record, which SFGetFile fills in before returning.

DESCRIPTION

The SFGetFile procedure displays a dialog box listing the names of a specific group of files from which the user can select one to be opened (as during an Open menu command). During the dialog, SFGetFile gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a file to open or by canceling the open operation. SFGetFile returns the user's input in a record of type SFReply.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for SFGetFile are

Trap macro	Selector
_Pack3	$0002

SPECIAL CONSIDERATIONS

Because SFGetFile may move memory, you should not call it at interrupt time.

SFPGetFile

Call the SFPGetFile procedure when your application requires more control over the Open dialog box than is possible using SFGetFile.

```
PROCEDURE SFPGetFile (where: Point; prompt: Str255;
                      fileFilter: FileFilterProcPtr;
                      numTypes: Integer; typeList: SFTypeList;
                      dlgHook: DlgHookProcPtr;
                      VAR reply: SFReply; dlgID: Integer;
                      filterProc: ModalFilterProcPtr);
```

where	The upper-left corner of the dialog box, in global coordinates.
prompt	Ignored.
fileFilter	A pointer to an optional file filter function, provided by your application, through which SFPGetFile passes files of the specified types.
numTypes	The number of file types to be displayed. If you specify a numTypes value of –1, the first filtering passes files of all types.
typeList	A list of file types to be displayed.
dlgHook	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of NIL if you have not added any items to the dialog box and want the standard items handled in the standard ways.
reply	The reply record, which SFPGetFile fills in before returning.
dlgID	The resource ID of a customized dialog template.
filterProc	A pointer to your modal-dialog filter function, which determines how the ModalDialog procedure filters events when called by the SFPGetFile procedure. Specify a value of NIL if you are not supplying your own function.

DESCRIPTION

The SFPGetFile procedure is an alternative to SFGetFile when you want to display a customized Open dialog box or handle the default dialog box in a customized way. During the dialog, SFPGetFile gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a file to open or by canceling the open operation. SFPGetFile returns the user's input in a record of type SFReply.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for SFPGetFile are

Trap macro	Selector
_Pack3	$0004

SPECIAL CONSIDERATIONS

Because SFPGetFile may move memory, you should not call it at interrupt time.

Application-Defined Routines

This section describes the application-defined routines whose addresses you pass to some of the Standard File Package routines. You can define

- a file filter function for determining which files the user can open
- a dialog hook function for handling user actions in the dialog boxes
- a modal-dialog filter function for handling user events received from the Event Manager
- an activation procedure for highlighting the display when keyboard input is directed at a customized field defined by your application

File Filter Functions

You specify a file filter function to determine which files appear in the displayed list of files and folders when the user is opening a file. You can define a standard or custom file filter.

MyStandardFileFilter

A file filter function whose address is passed to `StandardGetFile` should have the following form:

```
FUNCTION MyStandardFileFilter (pb: CInfoPBPtr): Boolean;
```

`pb` A pointer to a catalog information parameter block.

DESCRIPTION

When `StandardGetFile` is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application's specifications. If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

When your file filter function is called, it is passed, in the `pb` parameter, a pointer to a catalog information parameter block. See the chapter "File Manager" in this book for a description of the fields of this parameter block.

Your function evaluates the catalog information parameter block and returns a Boolean value that determines whether the file is filtered (that is, a value of `TRUE` suppresses display of the filename, and a value of `FALSE` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

SEE ALSO

See "Writing a File Filter Function" on page 3-20 for a sample file filter function.

MyCustomFileFilter

A file filter function whose address is passed to `CustomGetFile` should have the following form:

```
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr):
                                    Boolean;
```

`pb` A pointer to a catalog information parameter block.

`myDataPtr` A pointer to the optional data whose address is passed to `CustomGetFile`.

DESCRIPTION

When `CustomGetFile` is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application's specifications. If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

When your file filter function is called, it is passed, in the `pb` parameter, a pointer to a catalog information parameter block. See the chapter "File Manager" in this book for a description of the fields of this parameter block.

Your function evaluates the catalog information parameter block and returns a Boolean value that determines whether the file is filtered (that is, a value of `TRUE` suppresses display of the filename, and a value of `FALSE` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

SEE ALSO

See "Writing a File Filter Function" on page 3-20 for a sample file filter function.

Dialog Hook Functions

A dialog hook function handles user selections in a dialog box.

MyDlgHook

A dialog hook function should have the following form:

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr;
                    myDataPtr: Ptr): Integer;
```

`item` The number of the item selected.

`theDialog` A pointer to the dialog record of the dialog box.

`myDataPtr` A pointer to the optional data whose address is passed to `CustomGetFile` or `CustomPutFile`.

DESCRIPTION

You supply a dialog hook function to handle user selections of items that you added to a dialog box. If you provide a dialog hook function, `CustomPutFile` and `CustomGetFile` call your function immediately after calling `ModalDialog`. They pass your function the item number returned by `ModalDialog`, a pointer to the dialog record, and a pointer to the data received from your application, if any.

Your dialog hook function returns as its function result an integer that is either the item number passed to it or some other item number. If your dialog hook function does not handle a selection, it should pass the item number back to the Standard File Package for processing by setting its return value equal to the item number. If your dialog hook function does handle the selection, it should pass back `sfHookNullEvent` or the number of some other pseudo-item.

SEE ALSO

See "Writing a Dialog Hook Function" on page 3-21 for a sample dialog hook function.

Modal-Dialog Filter Functions

A modal-dialog filter function controls events closer to their source by filtering the events received from the Event Manager. The Standard File Package itself contains an internal modal-dialog filter function that maps keypresses and other user input onto the equivalent dialog box items. If you also want to process events at this level, you can supply your own filter function.

MyModalFilter

A modal-dialog filter function whose address is passed to `SFPGetFile` or `SFPPutFile` should have the following form:

```
FUNCTION MyModalFilter (theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
```

`theDialog`	A pointer to the dialog record of the dialog box.
`theEvent`	The event record for the event.
`itemHit`	The number of the item selected.

DESCRIPTION

Your modal-dialog filter function determines how the Dialog Manager procedure `ModalDialog` filters events. The `ModalDialog` procedure retrieves events by calling the Event Manager function `GetNextEvent`. The Standard File Package contains an internal filter function that performs some preliminary processing on each event it

receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `FALSE`, `ModalDialog` processes the event through its own filters. If your function returns a value of `TRUE`, `ModalDialog` returns with no further action.

SEE ALSO

See "Writing a Modal-Dialog Filter Function" on page 3-28 for a sample modal-dialog filter function.

MyModalFilterYD

A modal-dialog filter function whose address is passed to `CustomGetFile` or `CustomPutFile` should have the following form:

```
FUNCTION MyModalFilterYD (theDialog: DialogPtr;
                          VAR theEvent: EventRecord;
                          VAR itemHit: Integer;
                          myDataPtr: Ptr): Boolean;
```

`theDialog`	A pointer to the dialog record of the dialog box.
`theEvent`	The event record for the event.
`itemHit`	The number of the item selected.
`myDataPtr`	A pointer to the optional data whose address is passed to `CustomGetFile` or `CustomPutFile`.

DESCRIPTION

Your modal-dialog filter function determines how the Dialog Manager procedure `ModalDialog` filters events. The `ModalDialog` procedure retrieves events by calling the Event Manager function `GetNextEvent`. The Standard File Package contains an internal filter function that performs some preliminary processing on each event it receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `FALSE`, `ModalDialog` processes the event through its own filters. If your function returns a value of `TRUE`, `ModalDialog` returns with no further action.

SEE ALSO

See “Writing a Modal-Dialog Filter Function” on page 3-28 for a sample modal-dialog filter function.

Activation Procedures

An activation procedure controls the highlighting of dialog items that are defined by your application and can receive keyboard input.

MyActivateProc

An activation procedure should have the following form:

```
PROCEDURE MyActivateProc (theDialog: DialogPtr; itemNo: Integer;
                          activating: Boolean; myDataPtr: Ptr);
```

`theDialog` A pointer to the dialog record of the dialog box.

`itemNo` The number of the item selected.

`activating`
A Boolean value that specifies whether the field is being activated (`TRUE`) or deactivated (`FALSE`).

`myDataPtr` A pointer to the optional data whose address is passed to `CustomGetFile` or `CustomPutFile`.

DESCRIPTION

Your activation procedure controls the highlighting of dialog items that are defined by your application and can receive keyboard input. Ordinarily, you need to supply an activation procedure only if your application builds a list from which the user can select entries. The Standard File Package supplies the activation procedure for the file display list and for all TextEdit fields. You can also use the activation procedure to keep track of which field is receiving keyboard input, if your application needs that information.

Your application is responsible for removing the highlighting when one of its fields becomes inactive and for adding the highlighting when one of its fields becomes active. The Standard File Package can handle the highlighting of all TextEdit fields, even those defined by your application.

Summary of the Standard File Package

Pascal Summary

Constants

```
CONST
   {Gestalt selector and reply}
   gestaltStandardFileAttr =  'stdf';
   gestaltStandardFile58   =  0;

   {standard dialog resource IDs}
   sfPutDialogID           =  -6043;   {Save dialog box}
   sfGetDialogID           =  -6042;   {Open dialog box}

   {items that appear in both the Open and Save dialog boxes}
   sfItemOpenButton        =  1;       {Save or Open button}
   sfItemCancelButton      =  2;       {Cancel button}
   sfItemBalloonHelp       =  3;       {Balloon Help}
   sfItemVolumeUser        =  4;       {volume icon and name}
   sfItemEjectButton       =  5;       {Eject button}
   sfItemDesktopButton     =  6;       {Desktop button}
   sfItemFileListUser      =  7;       {display list}
   sfItemPopUpMenuUser     =  8;       {directory pop-up menu}
   sfItemDividerLinePict   =  9;       {dividing line between buttons}

   {items that appear in Save dialog boxes only}
   sfItemFileNameTextEdit  =  10;      {filename field}
   sfItemPromptStaticText  =  11;      {filename prompt text area}
   sfItemNewFolderUser     =  12;      {New Folder button}

   {pseudo-items available prior to version 7.0}
   sfHookFirstCall         =  -1;      {initialize display}
   sfHookCharOffset        =  $1000;   {offset for character input}
   sfHookNullEvent         =  100;     {null event}
   sfHookRebuildList       =  101;     {redisplay list}
   sfHookFolderPopUp       =  102;     {display parent-directory menu}
   sfHookOpenFolder        =  103;     {display contents of selected }
                                       { folder or volume}
```

```
{additional pseudo-items introduced in version 7.0}
sfHookLastCall          =  -2;      {clean up after display}
sfHookOpenAlias         =  104;     {resolve alias}
sfHookGoToDesktop       =  105;     {display contents of desktop}
sfHookGoToAliasTarget   =  106;     {select target of alias}
sfHookGoToParent        =  107;     {display contents of parent}
sfHookGoToNextDrive     =  108;     {display contents of next drive}
sfHookGoToPrevDrive     =  109;     {display contents of previous drive}
sfHookChangeSelection   =  110;     {select target of reply record}
sfHookSetActiveOffset   =  200;     {switch active item}

{refCon field in the window record in the dialog record}
sfMainDialogRefCon      =  'stdf';  {main dialog box}
sfNewFolderDialogRefCon =  'nfdr';  {New Folder dialog box}
sfReplaceDialogRefCon   =  'rplc';  {name conflict dialog box}
sfStatWarnDialogRefCon  =  'stat';  {stationery warning}
sfErrorDialogRefCon     =  'err ';  {general error report}
sfLockWarnDialogRefCon  =  'lock';  {software lock warning}

{resource IDs and item numbers of pre-7.0 dialog boxes}
putDlgID                =  -3999;   {Save dialog box}
putSave                 =  1;       {Save button}
putCancel               =  2;       {Cancel button}
putEject                =  5;       {Eject button}
putDrive                =  6;       {Drive button}
putName                 =  7;       {filename field}

getDlgID                =  -4000;   {Open dialog box}
getOpen                 =  1;       {Open button}
getCancel               =  3;       {Cancel button}
getEject                =  5;       {Eject button}
getDrive                =  6;       {Drive button}
getNmList               =  7;       {list of names}
getScroll               =  8;       {scroll bar}
```

Data Types

Standard File Reply Records

```
TYPE StandardFileReply  =       {enhanced standard file reply record}
   RECORD
      sfGood:        Boolean;    {TRUE if user did not cancel}
      sfReplacing:   Boolean;    {TRUE if replacing file with same name}
      sfType:        OSType;     {file type}
      sfFile:        FSSpec;     {selected file, folder, or volume}
      sfScript:      ScriptCode; {script of file, folder, or volume name}
      sfFlags:       Integer;    {Finder flags of selected item}
      sfIsFolder:    Boolean;    {selected item is a folder}
      sfIsVolume:    Boolean;    {selected item is a volume}
      sfReserved1:   LongInt;    {reserved}
      sfReserved2:   Integer;    {reserved}
   END;

   SFReply           =           {original standard file reply record}
   RECORD
      good:          Boolean;    {TRUE if user did not cancel}
      copy:          Boolean;    {reserved}
      fType:         OSType;     {file type}
      vRefNum:       Integer;    {working directory reference number}
      version:       Integer;    {reserved}
      fName:         Str63;      {filename}
   END;
```

Standard File Type List

```
   SFTypeList           =  ARRAY[0..3] OF OSType;
```

Callback Routine Pointer Types

```
   DlgHookProcPtr       =  ProcPtr;    {dialog hook function}
   DlgHookYDProcPtr     =  ProcPtr;    {dialog hook function with data}
   FileFilterProcPtr    =  ProcPtr;    {file filter function}
   FileFilterYDProcPtr  =  ProcPtr;    {file filter function with data}
   ModalFilterProcPtr   =  ProcPtr;    {modal-dialog filter}
   ModalFilterYDProcPtr =  ProcPtr;    {modal-dialog filter with data}
   ActivateYDProcPtr    =  ProcPtr;    {activation procedure}
```

Standard File Package Routines

Saving Files

```
PROCEDURE StandardPutFile    (prompt: Str255; defaultName: Str255;
                              VAR reply: StandardFileReply);
PROCEDURE CustomPutFile      (prompt: Str255; defaultName: Str255;
                              VAR reply: StandardFileReply; dlgID: Integer;
                              where: Point; dlgHook: DlgHookYDProcPtr;
                              filterProc: ModalFilterYDProcPtr;
                              activeList: Ptr;
                              activateProc: ActivateYDProcPtr;
                              yourDataPtr: UNIV Ptr);
PROCEDURE SFPutFile          (where: Point; prompt: Str255;
                              origName: Str255; dlgHook: DlgHookProcPtr;
                              VAR reply: SFReply);
PROCEDURE SFPPutFile         (where: Point; prompt: Str255;
                              origName: Str255; dlgHook: DlgHookProcPtr;
                              VAR reply: SFReply; dlgID: Integer;
                              filterProc: ModalFilterProcPtr);
```

Opening Files

```
PROCEDURE StandardGetFile    (fileFilter: FileFilterProcPtr;
                              numTypes: Integer; typeList: SFTypeList;
                              VAR reply: StandardFileReply);
PROCEDURE CustomGetFile      (fileFilter: FileFilterYDProcPtr;
                              numTypes: Integer; typeList: SFTypeList;
                              VAR reply: StandardFileReply; dlgID: Integer;
                              where: Point; dlgHook: DlgHookYDProcPtr;
                              filterProc: ModalFilterYDProcPtr;
                              activeList: Ptr;
                              activateProc: ActivateYDProcPtr;
                              yourDataPtr: UNIV Ptr);
PROCEDURE SFGetFile          (where: Point; prompt: Str255;
                              fileFilter: FileFilterProcPtr;
                              numTypes: Integer; typeList: SFTypeList;
                              dlgHook: DlgHookProcPtr; VAR reply: SFReply);
PROCEDURE SFPGetFile         (where: Point; prompt: Str255;
                              fileFilter: FileFilterProcPtr;
                              numTypes: Integer; typeList: SFTypeList;
                              dlgHook: DlgHookProcPtr; VAR reply: SFReply;
                              dlgID: Integer;
                              filterProc: ModalFilterProcPtr);
```

Application-Defined Routines

```
FUNCTION MyStandardFileFilter
                            (pb: CInfoPBPtr): Boolean;
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr): Boolean;
FUNCTION MyDlgHook          (item: Integer; theDialog: DialogPtr;
                             myDataPtr: Ptr): Integer;
FUNCTION MyModalFilter      (theDialog: DialogPtr;
                             VAR theEvent: EventRecord;
                             VAR itemHit: Integer): Boolean;
FUNCTION MyModalFilterYD    (theDialog: DialogPtr;
                             VAR theEvent: EventRecord;
                             VAR itemHit: Integer; myDataPtr: Ptr): Boolean;
PROCEDURE MyActivateProc    (theDialog: DialogPtr; itemNo: Integer;
                             activating: Boolean; myDataPtr: Ptr);
```

C Summary

Constants

```
/*Gestalt selector and reply*/
#define gestaltStandardFileAttr  'stdf'
#define gestaltStandardFile58    0

/*standard dialog resource IDs*/
enum {sfPutDialogID           = (-6043)}; /*Save dialog box*/
enum {sfGetDialogID           = (-6042)}; /*Open dialog box*/

/*items that appear in both the Open and Save dialog boxes/*
enum {sfItemOpenButton        = 1};    /*Save or Open button*/
enum {sfItemCancelButton      = 2};    /*Cancel button*/
enum {sfItemBalloonHelp       = 3};    /*Balloon Help*/
enum {sfItemVolumeUser        = 4};    /*volume icon and name*/
enum {sfItemEjectButton       = 5};    /*Eject button*/
enum {sfItemDesktopButton     = 6};    /*Desktop button*/
enum {sfItemFileListUser      = 7};    /*display list*/
enum {sfItemPopUpMenuUser     = 8};    /*directory pop-up menu*/
enum {sfItemDividerLinePict   = 9};    /*dividing line between buttons*/

/*items that appear in Save dialog boxes only*/
enum {sfItemFileNameTextEdit  = 10};   /*filename field*/
enum {sfItemPromptStaticText  = 11};   /*filename prompt text area*/
enum {sfItemNewFolderUser     = 12};   /*New Folder button*/
```

```
/*pseudo-items available prior to version 7.0*/
enum {sfHookFirstCall         = (-1)}; /*initialize display*/
enum {sfHookCharOffset        = 0x1000};/*offset for character input*/
enum {sfHookNullEvent         = 100};  /*null event*/
enum {sfHookRebuildList       = 101};  /*redisplay list*/
enum {sfHookFolderPopUp       = 102};  /*display parent-directory menu*/
enum {sfHookOpenFolder        = 103};  /*display contents of selected */
                                       /* folder or volume*/

/*additional pseudo-items introduced in version 7.0*/
enum {sfHookLastCall          = (-2)}; /*clean up after display*/
enum {sfHookOpenAlias         = 104};  /*resolve alias*/
enum {sfHookGoToDesktop       = 105};  /*display contents of desktop*/
enum {sfHookGoToAliasTarget   = 106};  /*select target of alias*/
enum {sfHookGoToParent        = 107};  /*display contents of parent*/
enum {sfHookGoToNextDrive     = 108};  /*display contents of next drive*/
enum {sfHookGoToPrevDrive     = 109};  /*display contents of previous drive*/
enum {sfHookChangeSelection   = 110};  /*select target of reply record*/
enum {sfHookSetActiveOffset   = 200};  /*switch active item*/

/*refCon field in the window record in the dialog record*/
#define sfMainDialogRefCon       'stdf'   /*main dialog box*/
#define sfNewFolderDialogRefCon  'nfdr'   /*New Folder dialog box*/
#define sfReplaceDialogRefCon    'rplc'   /*name conflict dialog box*/
#define sfStatWarnDialogRefCon   'stat'   /*stationery warning*/
#define sfErrorDialogRefCon      'err '   /*general error report*/
#define sfLockWarnDialogRefCon   'lock'   /*software lock warning*/

/*resource IDs and item numbers of pre-7.0 dialog boxes*/
enum {putDlgID                = -3999};   /*Save dialog box*/
enum {putSave                 = 1};       /*Save button*/
enum {putCancel               = 2};       /*Cancel button*/
enum {putEject                = 5};       /*Eject button*/
enum {putDrive                = 6};       /*Drive button*/
enum {putName                 = 7};       /*filename field*/

enum {getDlgID                = -4000};   /*Open dialog box*/
enum {getOpen                 = 1};       /*Open button*/
enum {getCancel               = 3};       /*Cancel button*/
enum {getEject                = 5};       /*Eject button*/
enum {getDrive                = 6};       /*Drive button*/
enum {getNmList               = 7};       /*list of names*/
enum {getScroll               = 8};       /*scroll bar*/
```

Data Types

Standard File Reply Records

```
struct StandardFileReply {       /*enhanced standard file reply record*/
      Boolean        sfGood;     /*TRUE if user did not cancel*/
      Boolean        sfReplacing;/*TRUE if replacing file with same name*/
      OSType         sfType;     /*file type*/
      FSSpec         sfFile;     /*selected file, folder, or volume*/
      ScriptCode     sfScript;   /*script of file, folder, or volume name*/
      short          sfFlags;    /*Finder flags of selected item*/
      Boolean        sfIsFolder; /*selected item is a folder*/
      Boolean        sfIsVolume; /*selected item is a volume*/
      long           sfReserved1;/*reserved*/
      short          sfReserved2;/*reserved*/
};

typedef struct StandardFileReply StandardFileReply;


struct SFReply {                 /*original standard file reply record*/
      Boolean        good;       /*TRUE if user did not cancel*/
      Boolean        copy;       /*reserved*/
      OSType         fType;      /*file type*/
      short          vRefNum;    /*working directory reference number*/
      short          version;    /*reserved*/
      Str63          fName;      /*filename*/
};

typedef struct SFReply SFReply;
```

Standard File Type List

```
typedef OSType SFTypeList[4];
```

Callback Routine Pointer Types

```
/*dialog hook function*/
typedef pascal short (*DlgHookProcPtr)
                           (short item, DialogPtr theDialog);
/*dialog hook function with data*/
typedef pascal short (*DlgHookYDProcPtr)
                           (short item, DialogPtr theDialog,
                            void *yourDataPtr);
```

```
/*file filter function*/
typedef pascal Boolean (*FileFilterProcPtr)
                            (ParmBlkPtr PB);
/*file filter function with data*/
typedef pascal Boolean (*FileFilterYDProcPtr)
                            (ParmBlkPtr PB, void *yourDataPtr);
/*modal-dialog filter*/
typedef pascal ProcPtr ModalFilterProcPtr;
                            (DialogPtr theDialog, EventRecord *theEvent,
                             short *itemHit);
/*modal-dialog filter with data*/
typedef pascal Boolean (*ModalFilterYDProcPtr)
                            (DialogPtr theDialog, EventRecord *theEvent,
                             short *itemHit, void *yourDataPtr);
/*activation procedure*/
typedef pascal void (*ActivateYDProcPtr)
                            (DialogPtr theDialog,
                             short itemNo, Boolean activating,
                             void *yourDataPtr);
```


Standard File Package Routines

Saving Files

```
pascal void StandardPutFile (const Str255 prompt, const Str255 defaultName,
                             StandardFileReply *reply);
pascal void CustomPutFile   (const Str255 prompt, const Str255 defaultName,
                             StandardFileReply *reply, short dlgID,
                             Point where, DlgHookYDProcPtr dlgHook,
                             ModalFilterYDProcPtr filterProc,
                             short *activeList,
                             ActivateYDProcPtr activateProc,
                             void *yourDataPtr);
pascal void SFPutFile       (Point where, const Str255 prompt,
                             const Str255 origName, DlgHookProcPtr dlgHook,
                             SFReply *reply);
pascal void SFPPutFile      (Point where, const Str255 prompt,
                             const Str255 origName, DlgHookProcPtr dlgHook,
                             SFReply *reply, short dlgID,
                             ModalFilterProcPtr filterProc);
```

Opening Files

```
pascal void StandardGetFile (const Str255 prompt,
                             FileFilterProcPtr fileFilter,
                             short numTypes, SFTypeList typeList,
                             StandardFileReply *reply);
pascal void CustomGetFile   (FileFilterYDProcPtr fileFilter,
                             short numTypes, SFTypeList typeList,
                             StandardFileReply *reply, short dlgID,
                             Point where, DlgHookYDProcPtr dlgHook,
                             ModalFilterYDProcPtr filterProc,
                             short *activeList,
                             ActivateYDProcPtr activateProc,
                             void *yourDataPtr);
pascal void SFGetFile       (Point where, const Str255 prompt,
                             FileFilterProcPtr fileFilter, short numTypes,
                             SFTypeList typeList, DlgHookProcPtr dlgHook,
                             SFReply *reply);
pascal void SFPGetFile      (Point where, const Str255 prompt,
                             FileFilterProcPtr fileFilter,
                             short numTypes, SFTypeList typeList,
                             DlgHookProcPtr dlgHook, SFReply *reply,
                             short dlgID, ModalFilterProcPtr filterProc);
```

Application-Defined Routines

```
pascal Boolean MyStandardFileFilter
                            (CInfoPBPtr pb);
pascal Boolean MyCustomFileFilter
                            (CInfoPBPtr pb, Ptr myDataPtr);
pascal short MyDlgHook      (short item, DialogPtr theDialog,
                             Ptr myDataPtr);
pascal Boolean MyModalFilter(DialogPtr theDialog,
                             EventRecord *theEvent, short *itemHit);
pascal Boolean MyModalFilterYD
                            (DialogPtr theDialog,
                             EventRecord *theEvent, short *itemHit,
                             Ptr myDataPtr);
pascal void MyActivateProc  (DialogPtr theDialog, short itemNo,
                             Boolean activating, Ptr myDataPtr);
```

Assembly-Language Summary

Data Structures

New Standard File Reply Record

0	sfGood	byte	command-valid flag
1	sfReplacing	byte	replace existing file flag
2	sfType	long	file type
6	sfFile	70 bytes	selected item
76	sfScript	word	display script
78	sfFlags	word	Finder flags from catalog
80	sfIsFolder	byte	folder flag
81	sfIsVolume	byte	volume flag
82	sfReserved1	long	reserved
86	sfReserved2	word	reserved

Old Standard File Reply Record

0	good	byte	command-valid flag
1	copy	byte	reserved
2	fType	long	file type
6	vRefNum	word	working directory reference number
8	version	word	reserved
10	fName	64 bytes	name of file (length byte followed by up to 63 characters)

Trap Macros

Trap Macro Requiring Routine Selector

```
_Pack3
```

Selector	Routine
$0001	SFPutFile
$0002	SFGetFile
$0003	SFPPutFile
$0004	SFPGetFile
$0005	StandardPutFile
$0006	StandardGetFile
$0007	CustomPutFile
$0008	CustomGetFile

Global Variables

CurDirStore	long	The directory ID of the current directory.
SFSaveDisk	word	The negative of the volume reference number of the current volume.

Alias Manager

Contents

This chapter describes how your application can use the Alias Manager to establish and resolve **alias records,** which are data structures that describe file system objects (that is, files, directories, and volumes). You create an alias record to take a "fingerprint" of a file system object, usually a file, that you might need to locate again later. You can store the alias record, instead of a file system specification, and then let the Alias Manager find the file again when it's needed. The Alias Manager contains algorithms for locating files that have been moved, renamed, copied, or restored from backup.

Note
The Alias Manager lets you manage alias records. It does not directly manipulate Finder aliases, which the user creates and manages through the Finder. The chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* describes Finder aliases and ways to accommodate them in your application. ◆

The Alias Manager is available only in system software version 7.0 or later. Use the `Gestalt` function, described in the chapter "Gestalt Manager" of *Inside Macintosh: Operating System Utilities*, to determine whether the Alias Manager is present.

Read this chapter if you want your application to create and resolve alias records. You might store an alias record, for example, to identify a customized dictionary from within a word-processing document. When the user runs a spelling checker on the document, your application can ask the Alias Manager to resolve the record to find the correct dictionary.

To use this chapter, you should be familiar with the File Manager's conventions for identifying files, directories, and volumes, as described in the chapter "Introduction to File Management" in this book.

This chapter begins with a description of the Alias Manager, alias records, and the search strategies that the Alias Manager uses to resolve an alias record. Then this chapter shows how you can

- create alias records
- resolve alias records
- store alias records as resources
- get information about the target of an alias record

About the Alias Manager

The Alias Manager is the part of the Operating System that helps you to locate specified files, directories, or volumes at a later time. It stores certain information about the **target** object in an alias record. When you later want the Alias Manager to find the target, you pass it the alias record and other information regarding the type of search to perform. Sometimes, the Alias Manager can find the original file. In other cases, the Alias Manager builds a list of potential matches and allows you (or the user) to select the desired file.

The Alias Manager creates and **resolves** (that is, finds the targets of) alias records. In general, you should use the Alias Manager to create an alias record whenever you find yourself storing a specific file description, such as filename and parent directory ID. The Alias Manager stores this information and more in the alias record, and it also provides a set of search strategies for resolving the record later. The search strategies are described in “Search Strategies” beginning on page 4-5. You can use the Alias Manager to create, resolve, and (if necessary) update alias records. You can also obtain information about the target of an alias record without actually resolving the alias record.

The Alias Manager can track files and directories across volumes. If the target of an alias record is on an unmounted AppleShare volume, the Alias Manager automatically mounts the volume when it resolves the alias. If the target object is on an unmounted ejectable volume, the Alias Manager prompts the user to insert the volume.

When the Alias Manager creates an alias record, it allocates the storage, fills in the record, and returns a handle to it. Your application is responsible for storing the record and retrieving it when needed. Your application must also supply strategies for handling various alias-resolution problems, described in “Resolving Alias Records” on page 4-10.

To help you understand and use the Alias Manager, this section provides

- an overview of alias records
- a description of the search strategies the Alias Manager uses to resolve alias records

Alias Records

An alias record is a data structure that describes a file, directory, or volume. The record contains

- location information, such as name and parent directory ID
- verification information, such as creation date, file type, and creator
- volume mounting information (that is, server and zone), if applicable

By storing alias records, you can allow your users to create a robust connection to a file—that is, a connection that can survive the moving or renaming of the target file. The Finder introduced in system software version 7.0, for example, stores alias records in aliases created by the user to represent other files or folders. The Edition Manager uses alias records to support data sharing among separate documents.

An alias record is a reliable way to identify a file system object when your application is communicating with a process that might be running on a different machine.

The creation of an alias record has no effect on the target of the record, except to establish a file ID reference for the target file if one did not previously exist. (See the chapter “File Manager” in this book for a description of file IDs and file ID references.)

The alias record contains only two fields of public information available to your application. The bulk of the record is managed privately by the Alias Manager.

```
TYPE AliasRecord =
   RECORD
      userType:   OSType;        {application's signature}
      aliasSize:  Integer;       {size of record when created}
      {variable-length private data}
   END;
```

Your application can store, in the `userType` field, its own signature or any other data that fits into 4 bytes. When the Alias Manager creates an alias record, it stores 0 in that field.

The Alias Manager stores, in the `aliasSize` field, the size assigned to the record at the time of its creation. Knowing the starting size allows you to store and retrieve data of your own at the end of the record (see “Customizing Alias Records” on page 4-13). An alias record is typically 200 to 300 bytes long.

The private Alias Manager data includes all of the location, verification, and mounting information needed to resolve the alias record with the various search strategies described in this chapter.

Search Strategies

Some of the key features of the Alias Manager are the search strategies built into the alias-resolution functions. The search strategies are designed to find the original target of an alias record, even if the target has been moved, renamed, copied, or restored from backup. Which strategy you use to resolve a particular alias record usually depends on a number of factors, including whether you are willing to sacrifice time to find as many potential targets as possible and whether the target is known to be in a particular volume. This section describes the available search strategies.

You can request either a relative or an absolute search. If you request an absolute search, you can specify whether the search should be either fast or exhaustive. (A relative search is always a fast search.) As you can see, there are three general search strategies available to your application for resolving alias records:

- relative search (always fast)
- absolute fast search
- absolute exhaustive search

The following sections describe these search strategies.

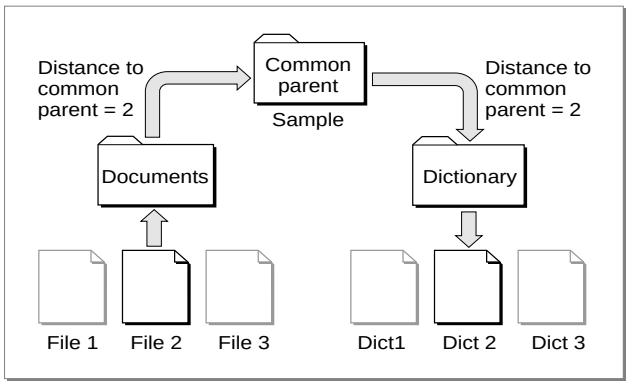
Relative Searches

During a **relative search,** the Alias Manager starts in a specified directory and searches for the target of an alias record by ascending the file system hierarchy to a predetermined common parent of the target and the starting directory and then descending the hierarchy from that common parent.

Suppose, for example, that you are writing a word-processing application that allows the user to build a customized, supplemental dictionary for each document. You might create the dictionary as a separate document in the same directory as the document it serves. In this case, the **common parent** of the document and the dictionary file (that is, the lowest-level directory that appears in the pathnames of both) is simply the directory containing both files.

More generally, you might want to store all document-specific dictionary files in their own directory, as illustrated in Figure 4-1. Here, the common parent of the document file "File 2" and its associated dictionary file "Dict 2" is the directory named "Sample."

Figure 4-1 Resolving a relative path



To resolve an alias record using a relative search, the Alias Manager needs several pieces of information, which are recorded in the alias record at the time you create it. The Alias Manager needs a **relative path,** that is, a path to the target from another file or directory on the same volume. (Relative paths don't work across volumes.) To record a relative path, the Alias Manager saves the distances from the target and the starting file or directory to their common parent. The Alias Manager can later use those distances in conjunction with the full pathname to conduct a relative search.

When resolving the alias record by using a relative path, the Alias Manager looks at the directory at the specified distance above the starting file or directory. The Alias Manager then constructs a partial pathname by extracting one field of the absolute pathname for each step from the target to the common parent. In Figure 4-1, the distance is 2, so the partial pathname is "Dictionary:Dict 2".

Absolute Searches

In contrast to a relative search, an **absolute search** always begins at the root directory of the file system hierarchy and always descends the hierarchy. The first step in any absolute search is to identify the volume on which the target resides. When conducting a volume search, the Alias Manager considers the volume's name, its creation date (which acts almost as a unique identifier for a volume), and its type (for example, a hard disk, a 3.5-inch floppy disk, or an AppleShare volume).

The Alias Manager first looks for a volume that matches all three criteria: name, creation date, and type. The search succeeds if the volume is mounted and if its name and creation date have not changed since the record was created. If the search fails, the Alias Manager attempts to match by creation date and type only. This step locates volumes that have been renamed. Finally, the Alias Manager attempts to match by volume name and type only.

If the target is on an unmounted AppleShare volume, the Alias Manager attempts to mount the volume. It presents a name and password dialog box if appropriate. If the target is on an unmounted ejectable volume, the Alias Manager displays a dialog box prompting the user to insert the volume. Your application can suppress the automatic mounting, as explained in the description of the `MatchAlias` function on page 4-20.

Note

Any time that your application needs to resolve a large number of aliases and the resolution of each alias might require user interaction, you should ensure that if the user cancels any of the dialog boxes, all remaining user interaction is canceled as well. ◆

In some circumstances, a relative search identifies the correct target when an absolute search cannot. For example, suppose the user of your word-processing application creates a working copy of a document and dictionary by copying the entire folder `Sample` to another disk. The user later updates the original document and dictionary by copying the folder from the working disk. All of the underlying file and directory identifications change, but the filenames and relative path remain the same. When the user later runs the spelling checker on the document, a relative-path search finds the correct target dictionary.

Fast Searches

A **fast search** employs an algorithm designed to find the target of an alias record quickly. Depending on how you invoke it, the fast-search algorithm starts with either a relative search or an absolute search. The Alias Manager can perform a relative fast search whether or not it has identified the target volume, but it cannot perform an absolute fast search unless the volume has been identified.

During an absolute fast search, the Alias Manager first searches by file ID (if the target is a file) or directory ID (if the target is a directory). (File IDs and directory IDs are described in the chapter “File Manager” in this book.) Even if a file has been renamed or moved on a volume, the Alias Manager can find it quickly through its file ID.

If the search by file ID or directory ID fails, the Alias Manager searches by name in the original parent directory. This search locates the target if its file or directory ID has changed but it still exists by the same name in the parent directory (for example, if the target was restored from a backup). The Alias Manager compares file numbers of files found by name in the correct parent directory. If the file numbers do not match, the file is treated as a possible match—that is, it is put on the list of candidates—and the search continues. If the target is not found by name in the parent directory, the Alias Manager looks for a file by file number in the parent directory. A file with the same file number but a different name replaces a file with the same name but a different file number in the list of matches.

If the search by file ID or directory ID fails and if the Alias Manager cannot find the original parent directory, it searches for the target by full pathname. This search succeeds if the target resides in the same location on the volume but the directory ID of its parent directory has changed (for example, if the entire parent directory was restored from a backup).

If the search by full pathname fails, the Alias Manager attempts to find the file by tracing partial pathnames up through all parent directories, using parent directory IDs instead of directory names. For example, consider this full pathname:

```
MyDisk:Fruits:Tropical:Ackees
```

If the search by full pathname fails, Alias Manager first looks for the partial pathname ":Ackees" in the directory with the ID that the directory "MyDisk:Fruits:Tropical" had when the alias record was created. If that search fails, it looks for ":Tropical:Ackees" in the directory with the ID that "MyDisk:Fruits" had, and so on.

If you do not ask for a search by relative path first but do provide a starting point for a relative search, and if the alias record contains relative path information, the Alias Manager performs a relative search after the absolute search. The relative search succeeds if the relative path is the same as when the record was created and if the names of the target and its intervening parent directories have not changed.

Exhaustive Searches

An **exhaustive search** uses an algorithm that scans an entire volume to look for possible matches. The Alias Manager typically performs an exhaustive search by calling the File Manager function `PBCatSearch`, searching for files or directories with a matching creation date, creator, and type. (See the chapter "File Manager" in this book for a description of `PBCatSearch`.)

The `PBCatSearch` function is available only on volumes that support the HFS routines and only on systems running system software version 7.0 and later. When `PBCatSearch` is not available, an exhaustive search of the entire volume is performed by making a series of indexed File Manager calls, searching for objects with matching creation date, type, creator, or file number.

Using the Alias Manager

You use the Alias Manager primarily to create and resolve alias records. You can also use it to get information about and update alias records.

The Alias Manager creates an alias record in memory and provides you with a handle to the record. When you no longer need a record in memory, free the memory by calling the Memory Manager's `DisposeHandle` procedure. Whenever possible, you should store and retrieve alias records as resources of type `'alis'`.

Alias Manager functions accept and return file specifications in the form of FSSpec records, which contain a volume reference number, a parent directory ID, and a target name. See the chapter "File Manager" in this book for a description of file identification conventions.

Before calling any of the Alias Manager functions, you should verify that the Alias Manager is available by calling the Gestalt function with a selector of gestaltAliasMgrAttr. If Gestalt sets the gestaltAliasMgrPresent bit in the response parameter, the Alias Manager is present.

For more detailed descriptions of the functions described in this section, see "Alias Manager Reference" beginning on page 4-13.

Creating Alias Records

You create a new alias record by calling one of three functions: NewAlias, NewAliasMinimal, or NewAliasMinimalFromFullPath. The NewAlias function creates a complete alias record that can make full use of the alias-resolution algorithms. The other two functions are streamlined variations designed for circumstances when speed is more important than robust resolution services. All three functions allocate the memory for the record, fill it in, and return a handle to it.

The NewAlias function always records the name and the file or directory ID of the target, its creation date, the parent directory name and ID, and the volume name and creation date. It also records the full pathname of the target and a collection of other information. You can have NewAlias store relative path information as well by supplying a starting point for a relative path (see "Relative Searches" on page 4-5 for a description of relative paths).

Call NewAlias when you want to create an alias record to store for later use. For example, suppose you are writing a word-processing application that allows the user to customize a dictionary for use with a single text file. Your application stores the custom data in a separate dictionary file in the same directory as the document. As soon as you create the dictionary file, you can call NewAlias to create an alias record for that file, including path information relative to the user's text file. Listing 4-1 shows how to use NewAlias to create a new alias.

Listing 4-1 Creating an alias record

```
FUNCTION DoCreateAlias (myDoc, myDict: FSSpec): OSErr;
VAR
   myAliasHdl: AliasHandle;                        {handle to created alias}
   myErr:      OSErr;
BEGIN
   myErr := NewAlias(@myDoc, myDict, myAliasHdl);  {create alias record}
   IF myAliasHdl <> NIL THEN
      myErr := DoSaveAlias(myDoc, myAliasHdl);     {save it as a resource}
   DoCreateAlias := myErr;                         {return result code}
END;
```

The function `DoCreateAlias` defined in Listing 4-1 takes two `FSSpec` records as parameters. The first specifies the document that is to serve as the starting point for a relative search, in this case the user's text file. The second `FSSpec` record specifies the target of the alias to be created, in this example the dictionary file. The `DoCreateAlias` function calls `NewAlias` to create the alias record; if successful, it calls the application-defined function `DoSaveAlias` to save the alias record as a resource in the document file's resource fork. See Listing 4-2 on page 4-12 for a definition of `DoSaveAlias`.

The two variations on the `NewAlias` function, `NewAliasMinimal` and `NewAliasMinimalFromFullPath`, record only a minimum of information about the target. The `NewAliasMinimal` function records only the target's name, parent directory ID, volume name and creation date, and volume mounting information. The `NewAliasMinimalFromFullPath` function records only the full pathname of the target, including the volume name.

Use `NewAliasMinimal` or `NewAliasMinimalFromFullPath` when you are willing to give up robust alias-resolution service in return for speed. The Finder, for example, stores minimal aliases in the Apple events that tell your application to open or print a document. Because the alias record is resolved almost immediately, the description is likely to remain valid, and the shorter record is probably safe.

You can use `NewAliasMinimalFromFullPath` to create an alias record for a target that doesn't exist or that resides on an unmounted volume.

Resolving Alias Records

The Alias Manager provides two functions that you can use to resolve alias records:

- the high-level function `ResolveAlias`, which performs a fast search and identifies only one target
- the low-level function `MatchAlias`, which can perform a fast search, an exhaustive search, or both and which can return a list of target candidates

In general, when you want to identify only the single most likely target of an alias record, you call `ResolveAlias`. You call `MatchAlias` when you want your program to control the search.

Identifying a Single Target

To resolve an alias record, you usually call the `ResolveAlias` function. This function performs a fast search (described earlier in "Fast Searches" on page 4-7) and exits after it identifies one target. The `ResolveAlias` function compares some key information about the identified target with the information stored in the alias record. If any of the information is different, `ResolveAlias` automatically updates the record.

Note

Like all other Alias Manager functions, `ResolveAlias` updates the record only in memory. Your application is responsible for updating alias records stored on disk when appropriate. ◆

In the dictionary example illustrated in Figure 4-1 on page 4-6, the application calls `ResolveAlias` with a relative path specification when the user runs the spelling checker on a document with a customized dictionary. If you provide a relative starting point, `ResolveAlias` performs the relative search first.

The `ResolveAlias` function reports, in the `wasChanged` parameter, whether it updated the alias record. After `ResolveAlias` runs, the value of `wasChanged` is `TRUE` if the record was updated and `FALSE` if it was not. If you are storing the alias record, check the value of `wasChanged` (as well as the function's result code) to see whether to update the stored record after resolving an alias.

If `ResolveAlias` can't resolve the alias record, it returns a nonzero result code. A result code of `fnfErr` signals that `ResolveAlias` has found the correct volume and parent directory but not the target file or folder. In this case, `ResolveAlias` constructs a valid `FSSpec` record that describes the target. You can use this record to explore possible solutions to the resolution failure. You can, for example, pass the `FSSpec` record to the File Manager function `FSpCreate` to create a replacement for a missing file.

Identifying Multiple Targets

The `MatchAlias` function is a low-level routine that gives your application control over the search algorithms.

You can control

- whether to attempt an automatic mounting of unmounted volumes
- whether to search on more than one volume
- whether to perform a fast search, an exhaustive search, or both
- what the order of the absolute and relative searches in a fast search should be
- whether to pursue search strategies that require interaction with the user (such as asking for a password while mounting an AppleShare volume)

You can also specify a maximum number of candidates that `MatchAlias` can identify. For details about controlling a search with the `MatchAlias` function, see its description beginning on page 4-20.

You can supply an optional filter function that `MatchAlias` calls

- each time it identifies a possible match
- when three seconds have elapsed without a match

The filter function determines whether each candidate is added to the list of possible targets. It can also terminate the search. See "Filtering Possible Targets" on page 4-25 for a description of the filter function.

The `MatchAlias` function returns, in an array of file system specification records, all candidates that it identifies.

Maintaining Alias Records

You can store alias records as resources of type `'alis'`.

```
CONST
   rAliasType = 'alis'; {resource type for saved alias records}
```

To store and retrieve resources, use the standard Resource Manager functions (`AddResource`, `GetResource`, and `GetNamedResource`) described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*. Listing 4-2 illustrates one way to save an alias record as a resource in a document file's resource fork.

Listing 4-2 Storing an alias record as a resource

```
FUNCTION DoSaveAlias (myDoc: FSSpec; myAliasHdl: AliasHandle): OSErr;
VAR
   myErr: OSErr;
   myFile: Integer;           {file ref number of document's resource fork}
CONST
   kID = 129;
   kName = 'Dictionary Alias';
BEGIN
   myFile := FSpOpenResFile(myDoc, fsCurPerm);
   IF myFile = -1 THEN        {couldn't open the document's resource fork}
      BEGIN
         DoSaveAlias := ResError;
         exit(DoSaveAlias);
      END;
   AddResource(Handle(myAliasHdl), rAliasType, kID, kName);
   myErr := ResError;         {check for errors adding resource}
   IF myErr = noErr THEN
      BEGIN
         WriteResource(Handle(myAliasHdl));
         myErr := ResError;   {check for errors writing resource}
      END;
   DoSaveAlias := myErr;
END;
```

Note that `DoSaveAlias` assumes that the file specified by the `myDoc` parameter already has a resource fork and that the file is not yet open. Your application might have different requirements.

To update an alias record, use the `UpdateAlias` function. You typically call `UpdateAlias` any time you know that the target of an alias record has been renamed or otherwise changed. You are most likely to call `UpdateAlias` after a call to the

`MatchAlias` function. If `MatchAlias` identifies a single target, it sets a flag telling you whether or not the key information about the target file matches the information in the alias record. It is the responsibility of your application to update the record.

The `ResolveAlias` function automatically updates an alias record if any of the key information about the identified target does not match the information in the record.

Getting Information From Alias Records

To retrieve information from an alias record without actually resolving the record, call the `GetAliasInfo` function. You can use `GetAliasInfo` to retrieve the name of the target, the names of the target's parent directories, the name of the target's volume, or, in the case of an AppleShare volume, its zone or server name.

The information returned by `GetAliasInfo` might be stale. `GetAliasInfo` reads the information stored in the alias record, which might have changed since the creation of the record. Because it doesn't resolve the alias record, `GetAliasInfo` is most useful for providing information quickly.

Customizing Alias Records

An alias record contains two kinds of information: public information available to your application and private information available only to the Alias Manager. Your application can use the first field, `userType`, to store its own signature or any other data that fits into 4 bytes. Your application can use the second field, `aliasSize`, to customize the alias record for storing additional data.

The Alias Manager stores, in the `aliasSize` field, the size of the record at the time it is created or updated. To customize an alias record, you first use the Memory Manager's `SetHandleSize` procedure to increase the size of the record. You can then find the starting address of your own data in the record by adding the record's starting address to the length recorded in the `aliasSize` field. If you use the Memory Manager to expand the record, the Alias Manager preserves your data, even if it changes the size of its own data when updating the record.

Note
In general, you should customize only alias records that you have created. ◆

Alias Manager Reference

This section describes the routines provided by the Alias Manager and the `AliasRecord` data structure you must pass when calling those routines.

Data Structures

The Alias Manager uses alias records to store information that allows it to locate an object in the file system.

Alias Records

Alias records are defined by the `AliasRecord` data type.

```
TYPE AliasRecord  =
RECORD
   userType:   OSType;        {application's signature}
   aliasSize:  Integer;       {size of record when created}
   {variable-length private data}
END;
```

Field descriptions

`userType` A 4-byte field that can contain application-specific data. When an alias record is created, this field contains 0. Your application can use this field for its own purposes. Typically you should store your application's signature here.

`aliasSize` The size, in bytes, assigned to the alias record at the time of its creation or updating. This is the total size of the record, including the `userType` and `aliasSize` fields, as well as the variable-length data that is private to the Alias Manager.

Following these two fields is a variable-length block of data maintained privately by the Alias Manager.

Alias Manager Routines

This section describes the routines you use to create, update, resolve, and read alias records. Alias Manager routines use file system specification records (defined by the `FSSpec` data type) to identify files, directories, and volumes. To create an `FSSpec` record, call the function `FSMakeFSSpec`, described in the chapter "File Manager" in this book.

The Alias Manager routines can return the result codes listed in this section or any other applicable file system or memory management result codes.

Creating and Updating Alias Records

You can use the functions `NewAlias`, `NewAliasMinimal`, `NewAliasMinimalFromFullPath`, and `UpdateAlias` to create and update alias records.

NewAlias

You use the `NewAlias` function to create a complete alias record.

```
FUNCTION NewAlias (fromFile: FSSpecPtr; target: FSSpec;
                   VAR alias: AliasHandle): OSErr;
```

`fromFile` The starting point for a relative path, to be used later in a relative search. If you do not need relative path information in the record, pass a `fromFile` value of `NIL`. If you want `NewAlias` to record relative path information, pass a pointer to a valid `FSSpec` record in this parameter. The two files or directories, `fromFile` and `target`, must reside on the same volume.

`target` An `FSSpec` record for the target of the alias record.

`alias` A handle to the newly created alias record. If the function fails to create an alias record, it sets `alias` to `NIL`.

DESCRIPTION

The `NewAlias` function creates an alias record that describes the specified target. It allocates the storage, fills in the record, and puts a record handle to that storage in the `alias` parameter. `NewAlias` always records the name and file or directory ID of the target, its creation date, the parent directory name and ID, and the volume name and creation date. It also records the full pathname of the target and a collection of other information relevant to locating the target, verifying the target, and mounting the target's volume, if necessary. You can have `NewAlias` store relative path information as well by supplying a starting point for a relative path (see "Relative Searches" on page 4-5 for a description of relative paths).

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `NewAlias` are

Trap macro	Selector
`_AliasDispatch`	$0002

RESULT CODE

`noErr`	0	No error

NewAliasMinimal

You use the `NewAliasMinimal` function to create a short alias record quickly.

```
FUNCTION NewAliasMinimal (target: FSSpec;
                          VAR alias: AliasHandle): OSErr;
```

`target` An `FSSpec` record for the target of the alias record.

`alias` A handle to the newly created alias record. If the function fails to create an alias record, it sets `alias` to `NIL`.

DESCRIPTION

The `NewAliasMinimal` function creates an alias record that contains only the minimum information necessary to describe the target: the target name, the parent directory ID, the volume name and creation date, and the volume mounting information. The `NewAliasMinimal` function uses the standard alias record data structure, but it fills in only parts of the record.

Note
The `ResolveAlias` function, described on page 4-19, never updates a minimal alias record. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `NewAliasMinimal` are

Trap macro	Selector
`_AliasDispatch`	$0008

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	The value of `target` or `alias` parameter, or of both, is `NIL`, or the alias record is corrupt

NewAliasMinimalFromFullPath

You use the function `NewAliasMinimalFromFullPath` to quickly create an alias record that contains only the full pathname of the target.

```
FUNCTION NewAliasMinimalFromFullPath
                  (fullPathLength: Integer; fullPath: Ptr;
                     zoneName: Str32; serverName: Str31;
                     VAR alias: AliasHandle): OSErr;
```

`fullPathLength`
The number of characters in the full pathname of the target.

`fullPath` A pointer to a buffer that contains the full pathname of the target. The full pathname starts with the name of the volume, includes all of the directory names in the path to the target, and ends with the target name. (For a description of pathnames, see the chapter "File Manager" in this book.)

`zoneName` The AppleTalk zone name of the AppleShare volume on which the target resides. Set this parameter to a null string if you do not need it.

`serverName` The AppleTalk server name of the AppleShare volume on which the target resides. Set this parameter to a null string if you do not need it.

`alias` A handle to the newly created alias record. If the function fails to create an alias record, it sets `alias` to `NIL`.

DESCRIPTION

The `NewAliasMinimalFromFullPath` function creates an alias record that identifies the target by full pathname. You can call `NewAliasMinimalFromFullPath` to create an alias record for a file that doesn't exist or that resides on an unmounted volume.

The `NewAliasMinimalFromFullPath` function uses the standard alias record data structure, but it fills in only the information provided in the input parameters. You can therefore use `NewAliasMinimalFromFullPath` to create alias records for targets on unmounted volumes.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `NewAliasMinimalFromFullPath` are

Trap macro	Selector
`_AliasDispatch`	$0009

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter error

UpdateAlias

You use the `UpdateAlias` function to update an alias record.

```
FUNCTION UpdateAlias (fromFile: FSSpecPtr; target: FSSpec;
                      alias: AliasHandle;
                      VAR wasChanged: Boolean): OSErr;
```

`fromFile` The starting point for a relative path, to be used later in a relative search. If you do not need relative path information in the record, pass a `fromFile` value of `NIL`. If you want `UpdateAlias` to record relative path information, pass a pointer to a valid `FSSpec` record in this parameter.

`target` The target of the alias record. This parameter must be a valid `FSSpec` record.

`alias` A handle to the alias record to be updated.

`wasChanged` A Boolean value indicating whether the newly constructed alias record is exactly the same as the old one. If the new record is the same as the old one, `UpdateAlias` sets the `wasChanged` parameter to `FALSE`. Otherwise, it sets it to `TRUE`. Check this parameter to determine whether you need to save an updated record.

DESCRIPTION

The `UpdateAlias` function updates the alias record pointed to by the `alias` parameter so that it describes the target specified by the `target` parameter. The `UpdateAlias` function rebuilds the entire alias record and fills it in as the `NewAlias` function would.

The `UpdateAlias` function always creates a complete alias record. When you use `UpdateAlias` to update a minimal alias record, you convert the minimal record to a complete record.

SPECIAL CONSIDERATIONS

The two files or directories, `fromFile` and `target`, must reside on the same volume.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `UpdateAlias` are

Trap macro	Selector
`_AliasDispatch`	$0006

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	The value of the `target` or `alias` parameter, or both, is `NIL`, or the alias record is corrupt

Resolving and Reading Alias Records

You can use the functions `ResolveAlias` and `MatchAlias` to resolve or find possible targets of an alias record. You can use the function `GetAliasInfo` to get information about the target of an alias without actually resolving the alias.

ResolveAlias

You use the `ResolveAlias` function to identify the single most likely target of an alias record.

```
FUNCTION ResolveAlias (fromFile: FSSpecPtr; alias: AliasHandle;
                       VAR target: FSSpec;
                       VAR wasChanged: Boolean): OSErr;
```

`fromFile` The starting point for a relative search. If you pass a `fromFile` parameter of `NIL`, `ResolveAlias` performs only an absolute search. If you pass a pointer to a valid `FSSpec` record in the `fromFile` parameter, `ResolveAlias` performs a relative search for the target, followed by an absolute search only if the relative search fails. If you want to perform an absolute search followed by a relative search, you must use the `MatchAlias` function.

`alias` A handle to the alias record to be resolved and, if necessary, updated.

`target` The target of the alias record. This parameter must be a valid `FSSpec` record.

`wasChanged` A Boolean value indicating whether the alias record to be resolved was updated because it contained some outdated information about the target.

DESCRIPTION

The `ResolveAlias` function performs a fast search for the target of the alias, as described in “Fast Searches” on page 4-7. If the resolution is successful, `ResolveAlias` returns (in the `target` parameter) the `FSSpec` record for the target file system object, updates the alias record if necessary, and reports (through the `wasChanged` parameter) whether the record was updated. If the target is on an unmounted AppleShare volume, `ResolveAlias` automatically mounts the volume. If the target is on an unmounted ejectable volume, `ResolveAlias` asks the user to insert the volume. The `ResolveAlias` function exits after it finds one acceptable target.

After it identifies a target, `ResolveAlias` compares some key information about the target with the information in the alias record. (The description of the `MatchAlias` function, beginning on page 4-20, lists the key information.) If the information differs, `ResolveAlias` updates the record to match the target. If it updates the alias record, `ResolveAlias` sets the `wasChanged` parameter to `TRUE`. Otherwise, it sets it to `FALSE`. (`ResolveAlias` never updates a minimal alias, so it never sets `wasChanged` to `TRUE` when resolving a minimal alias.)

When it finds the specified volume and parent directory but fails to find the target file or directory in that location, `ResolveAlias` returns a result code of `fnfErr` and fills in the `target` parameter with a complete `FSSpec` record describing the target (that is, the volume reference number, parent directory ID, and filename or folder name). The `FSSpec` record is valid, although the object it describes does not exist. This information is intended as a "hint" that lets you explore possible solutions to the resolution failure. You can, for example, pass the `FSSpec` record to the File Manager function `FSpCreate` to create a replacement for a missing file.

The `ResolveAlias` function displays the standard dialog boxes when it needs input from the user, such as a name and password for mounting a remote volume. The user can cancel the resolution through these dialog boxes.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `ResolveAlias` are

Trap macro	Selector
`_AliasDispatch`	$0003

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	The volume is not mounted
`fnfErr`	–43	Target not found, but volume and parent directory found
`paramErr`	–50	The value of the `target` or `alias` parameter, or both, is `NIL`, or the alias record is corrupt
`dirNFErr`	–120	Parent directory not found
`usrCanceledErr`	–128	The user canceled the operation

MatchAlias

You use the `MatchAlias` function to identify a list of possible matches and pass the list through an optional selection filter. The filter can return more than one possible match.

```
FUNCTION MatchAlias (fromFile: FSSpecPtr; rulesMask: LongInt;
                     alias: AliasHandle; VAR aliasCount: Integer;
                     aliasList: FSSpecArrayPtr;
                     VAR needsUpdate: Boolean;
                     aliasFilter: AliasFilterProcPtr;
                     yourDataPtr: UNIV Ptr): OSErr;
```

`fromFile` The starting point for a relative search. If you do not want `MatchAlias` to perform a relative search, set `fromFile` to `NIL`. If you want `MatchAlias` to perform a relative search, pass a pointer to a file system specification record that describes the starting point for the search.

`rulesMask` A set of rules to guide the resolution. Pass the sum of all of the rules you want to invoke.

`alias` A handle to the alias record to be resolved.

`aliasCount` On input, the maximum number of possible matches to return. On output, the actual number of matches returned.

`aliasList` A pointer to the array that holds the results of the search.

`needsUpdate`
A Boolean flag that indicates whether the alias record to be resolved needs to be updated.

`aliasFilter`
An application-defined filter function.

`yourDataPtr`
A pointer to data to be passed to the filter function.

DESCRIPTION

The `MatchAlias` function resolves the alias record specified by the `alias` parameter, following the rules specified by the `rulesMask` parameter. Then it returns, in the structure specified by the `aliasList` parameter, a list of possible candidates. The `MatchAlias` function places, in the `aliasCount` parameter, the number of candidates identified.

You specify the matching criteria by passing a sum of these constants in the `rulesMask` parameter.

```
CONST
   kARMMountVol         =  $00000001;{mount volume automatically}
   kARMNoUI             =  $00000002;{suppress user interface}
   kARMMultVols         =  $00000008;{search on multiple volumes}
   kARMSearch           =  $00000100;{do a fast search}
   kARMSearchMore       =  $00000200;{do an exhaustive search}
   kARMSearchRelFirst   =  $00000400;{do a relative search first}
```

Constant descriptions

`kARMMountVol` Automatically try to mount the target's volume if it is not mounted.

`kARMNoUI` Stop if a search requires user interaction, such as a password dialog box when mounting a remote volume. If user interaction is needed and `kARMNoUI` is in effect, the search fails.

`kARMMultVols` Search all mounted volumes. The search begins with the volume on which the target resided when the record was created. When you specify a fast search of all mounted volumes, `MatchAlias` performs a formal fast search only on the volume described in the alias record. On all other volumes it looks for the target by ID or by name in the directory with the specified parent directory ID. When you specify an exhaustive search of multiple volumes, `MatchAlias` performs the same search on all volumes. When resolving an alias record created by `NewAliasMinimalFromFullPath`, `MatchAlias` ignores this flag.

`kARMSearch` Perform a fast search for the alias target. If `kARMSearchRelFirst` is not set, perform an absolute search first, followed by a relative search only if the value of the `fromFile` parameter is not `NIL` and the list of matches is not full.

`kARMSearchMore` Perform an exhaustive search for the alias target. On HFS volumes, the exhaustive search uses the File Manager function `PBCatSearch` to identify candidates with matching creation date, type, and creator. The `PBCatSearch` function is available only on HFS volumes and only on systems running version 7.0 or later. On MFS volumes or HFS volumes that do not support `PBCatSearch`, the exhaustive search makes a series of indexed calls to File Manager functions, using the same search criteria. If you set `kARMSearchMore` and either or both of `kARMSearch` and `kARMSearchRelFirst`, `MatchAlias` performs the fast search first.

`kARMSearchRelFirst`
If `kARMSearch` is also set, perform a relative search before the absolute search. (If `kARMSearch` is also set and the target is found through the absolute search, `MatchAlias` sets the `needsUpdate` flag to `TRUE`.) If neither `kARMSearch` nor `kARMSearchMore` is set, perform only a relative search. If `kARMSearch` is not set but `kARMSearchMore` is set, perform a relative search followed by an exhaustive search.

You must specify at least one of the last three parameters: `kARMSearch`, `kARMSearchMore`, and `kARMSearchRelFirst`.

Your application can specify a maximum number of possible matches by setting the `aliasCount` parameter. `MatchAlias` changes the `aliasCount` parameter to the actual number of candidates identified. If `MatchAlias` finds the parent directory on the correct volume but does not find the target, it sets the `aliasCount` parameter to 1, puts the file system specification record for the target in the results list, and returns `fnfErr`. The `FSSpec` record is valid, although the object it describes does not exist. This information is intended as a “hint” that lets you explore possible solutions to the resolution failure. You can, for example, use the `FSSpec` record and the File Manager function `FSpCreate` to create a replacement for a missing file.

The `needsUpdate` flag is a signal to your application that the record might need to be updated. After it identifies a target, `MatchAlias` compares some key information about the target with the same information in the record. If the information does not match, `MatchAlias` sets the `needsUpdate` flag to `TRUE`. The key information is

- the name of the target
- the directory ID of the target’s parent
- the file ID or directory ID of the target
- the name and creation date of the volume on which the target resides

The `MatchAlias` function also sets the `needsUpdate` flag to `TRUE` if it identifies a list of possible matches rather than a single match or if `kARMsearchRelFirst` is set but the target is identified through either an absolute search or an exhaustive search. Otherwise, the `MatchAlias` function sets the `needsUpdate` flag to `FALSE`. `MatchAlias` always

sets the `needsUpdate` flag to `FALSE` when resolving an alias created by `NewAliasMinimal`. If you want to update the alias record to reflect the final results of the resolution, call `UpdateAlias`.

The `aliasFilter` parameter points to a filter function supplied by your application. The Alias Manager executes this function each time it identifies a possible match and after the search has continued for three seconds without a match. Your filter function returns a Boolean value that determines whether the possible match is discarded (`TRUE`) or added to the list of possible targets (`FALSE`). It can also terminate the search by setting the variable parameter `quitFlag`. See "Filtering Possible Targets" on page 4-25 for a description of the filter function.

The `yourDataPtr` parameter can point to any data that your application might need in the filter function.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `MatchAlias` are

Trap macro	Selector
`_AliasDispatch`	$0005

RESULT CODES

`noErr`	0	No error
`nsvErr`	–35	The volume is not mounted
`fnfErr`	–43	Target not found, but volume and parent directory found
`paramErr`	–50	The value of the `target` or `alias` parameter, or both, is `NIL`, or the alias record is corrupt
`usrCanceledErr`	–128	The user canceled the operation

GetAliasInfo

You use the `GetAliasInfo` function to get information from an alias record without actually resolving the record.

```
FUNCTION GetAliasInfo (alias: AliasHandle; index: AliasInfoType;
                       VAR theString: Str63): OSErr;
```

`alias`	A handle to the alias record to be read.
`index`	The kind of information to be retrieved.
`theString`	A string that holds the requested information.

DESCRIPTION

The `GetAliasInfo` function retrieves the information specified by the `index` parameter from the record pointed to by the `alias` parameter and places that information in the parameter `theString`.

The `index` parameter specifies the kind of information to be retrieved. If the value of `index` is a positive integer, `GetAliasInfo` retrieves the parent directory that has the same hierarchical level above the target as the index parameter (for example, an `index` value of 2 returns the name of the parent directory of the target's parent directory). You can therefore assemble the names of the target and all of its parent directories by making repeated calls to `GetAliasInfo` with incrementing index values, starting with a value of 0. When the value of `index` is greater than the number of levels between the target and the root, `GetAliasInfo` returns an empty string. You can also set the `index` parameter to one of the following five values:

```
CONST
   asiZoneName          =  -3;      {get zone name}
   asiServerName        =  -2;      {get server name}
   asiVolumeName        =  -1;      {get volume name}
   asiAliasName         =  0;       {get target name}
   asiParentName        =  1;       {get parent directory name}
```

Constant descriptions

`asiZoneName`	If the record represents a target on an AppleShare volume, retrieve the server's zone name. Otherwise, return an empty string.
`asiServerName`	If the record represents a target on an AppleShare volume, retrieve the server name. Otherwise, return an empty string.
`asiVolumeName`	Return the name of the volume on which the target resides.
`asiAliasName`	Return the name of the target.
`asiParentName`	Return the name of the parent directory of the target of the record. If the target is a volume, return the volume name.

The `GetAliasInfo` function returns the information stored in the alias record, which might not be current. To ensure that the information is current, you can resolve and update the alias record before calling `GetAliasInfo`.

Note

The `GetAliasInfo` function cannot provide all kinds of information about a minimal alias. ◆

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `GetAliasInfo` are

Trap macro	Selector
`_AliasDispatch`	$0007

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	The value of `alias` or `theString` parameter, or both, is `NIL`; the value of `index` is less than the value of `asiZoneName`; or the alias record is corrupt

Application-Defined Routines

The Alias Manager supports a single application-defined routine, a function for filtering out possible targets of an alias record.

Filtering Possible Targets

You can write your own filter function to examine possible targets identified by the `MatchAlias` function. The `MatchAlias` function calls your filter function each time it identifies a possible match or when three seconds have elapsed without a match.

MyMatchAliasFilter

You can pass the address of an alias-matching filter function to the `MatchAlias` function.

```
FUNCTION MyMatchAliasFilter (cpbPtr: CInfoPBPtr;
                             VAR quitFlag: Boolean;
                             myDataPtr: Ptr): Boolean;
```

`cpbPtr`	A pointer to a catalog information parameter block.
`quitFlag`	On exit, set this to `TRUE` if you want to terminate the search.
`myDataPtr`	A pointer to custom data.

DESCRIPTION

Your application-defined filter function is called by `MatchAlias` to filter out possible matches. When your function is called, the `cpbPtr` parameter points to the catalog information parameter block of the possible match (returned by the File Manager function `PBGetCatInfo`). The `MatchAlias` function sets this parameter to `NIL` if it is calling your function to give it the periodic chance to terminate the search. (Do not use this pointer without checking for `NIL`.) If you want to terminate the search, set the `quitFlag` parameter to `TRUE`.

The `myDataPtr` parameter points to any customized data that your application passed when it called `MatchAlias`. This parameter allows your filter function to access any data that your application has set up on its own.

Your function should return `TRUE` to indicate that the possible match is to be discarded, or `FALSE` to indicate that the possible match is to be added to the list of possible targets.

Summary of the Alias Manager

Pascal Summary

Constants

```
CONST
   {Gestalt constants}
   gestaltAliasMgrAttr     = 'alis';      {Alias Mgr attributes selector}
   gestaltAliasMgrPresent  = 0;           {Alias Mgr is present}

   {resource type for saved alias records}
   rAliasType              = 'alis';

   {masks for alias resolution action rules used by MatchAlias}
   kARMMountVol            =  $00000001;  {mount volume automatically}
   kARMNoUI                =  $00000002;  {suppress user interface}
   kARMMultVols            =  $00000008;  {search on multiple volumes}
   kARMSearch              =  $00000100;  {do a fast search}
   kARMSearchMore          =  $00000200;  {do an exhaustive search}
   kARMSearchRelFirst      =  $00000400;  {do a relative search first}

   {index values for GetAliasInfo}
   asiZoneName             =  –3;         {get zone name}
   asiServerName           =  –2;         {get server name}
   asiVolumeName           =  –1;         {get volume name}
   asiAliasName            =  0;          {get target name}
   asiParentName           =  1;          {get parent directory name}
```

Data Types

```
TYPE AliasRecord     =                {alias record}
   RECORD
      userType:   OSType;             {application's signature}
      aliasSize:  Integer;            {size of record when created}
      {variable-length private data}
   END;
```

```
AliasPtr            =  ^AliasRecord;
AliasHandle         =  ^AliasPtr;

AliasInfoType       =  Integer;    {alias record information type}

AliasFilterProcPtr  =  ProcPtr;    {application-defined routine}
```

Alias Manager Routines

Creating and Updating Alias Records

```
FUNCTION NewAlias           (fromFile: FSSpecPtr; target: FSSpec;
                             VAR alias: AliasHandle): OSErr;
FUNCTION NewAliasMinimal    (target: FSSpec;
                             VAR alias: AliasHandle): OSErr;
FUNCTION NewAliasMinimalFromFullPath
                            (fullPathLength: Integer; fullPath: Ptr;
                             zoneName: Str32; serverName: Str31;
                             VAR alias: AliasHandle): OSErr;
FUNCTION UpdateAlias        (fromFile: FSSpecPtr; target: FSSpec;
                             alias: AliasHandle;
                             VAR wasChanged: Boolean): OSErr;
```

Resolving and Reading Alias Records

```
FUNCTION ResolveAlias       (fromFile: FSSpecPtr; alias: AliasHandle;
                             VAR target: FSSpec;
                             VAR wasChanged: Boolean): OSErr;
FUNCTION MatchAlias         (fromFile: FSSpecPtr; rulesMask: LongInt;
                             alias: AliasHandle; VAR aliasCount: Integer;
                             aliasList: FSSpecArrayPtr;
                             VAR needsUpdate: Boolean;
                             aliasFilter: AliasFilterProcPtr;
                             yourDataPtr: UNIV Ptr): OSErr;
FUNCTION GetAliasInfo       (alias: AliasHandle; index: AliasInfoType;
                             VAR theString: Str63): OSErr;
```

Application-Defined Routine

```
FUNCTION MyMatchAliasFilter (cpbPtr: CInfoPBPtr; VAR quitFlag: Boolean;
                             myDataPtr: Ptr): Boolean;
```

C Summary

Constants

```
/*Gestalt constants*/
#define gestaltAliasMgrAttr   'alis'      /*Alias Mgr attributes selector*/
#define gestaltAliasMgrPresent 0          /*Alias Mgr is present*/

/*resource type for saved alias records*/
#define rAliasType            'alis'

/*masks for alias resolution action rules used by MatchAlias*/
enum {kARMMountVol      = 0x00000001};    /*mount volume automatically*/
enum {kARMNoUI          = 0x00000002};    /*suppress user interface*/
enum {kARMMultVols      = 0x00000008};    /*search on multiple volumes*/
enum {kARMSearch        = 0x00000100};    /*do a fast search*/
enum {kARMSearchMore    = 0x00000200};    /*do an exhaustive search*/
enum {kARMSearchRelFirt = 0x00000400};    /*do a relative search first*/

/*index values for GetAliasInfo*/
enum {asiZoneName       = –3};            /*get zone name*/
enum {asiServerName     = –2};            /*get server name*/
enum {asiVolumeName     = –1};            /*get volume name*/
enum {asiAliasName      = 0};             /*get target name*/
enum {asiParentName     = 1};             /*get parent directory name*/
```

Data Types

```
typedef struct {                          /*alias record*/
      OSType            userType;         /*application's signature*/
      unsigned short    aliasSize;        /*size of record when created*/
} AliasRecord;

typedef AliasRecord *AliasPtr;
typedef AliasRecord **AliasHandle;
typedef short AliasInfoType;              /*alias record information type*/

typedef pascal Boolean (*AliasFilterProcPtr)(CInfoPBPtr cpbPtr,
                        Boolean *quitFlag, Ptr yourDataPtr);
```

Alias Manager Routines

Creating and Updating Alias Records

```
pascal OSErr NewAlias       (const FSSpec *fromFile, const FSSpec *target,
                             AliasHandle *alias);
pascal OSErr NewAliasMinimal(const FSSpec *target, AliasHandle *alias);
pascal OSErr NewAliasMinimalFromFullPath
                            (short fullPathLength,
                             const unsigned char *fullpath,
                             const Str32 zoneName, const Str31 serverName,
                             AliasHandle *alias);
pascal OSErr UpdateAlias    (const FSSpec *fromFile, const FSSpec *target,
                             AliasHandle alias, Boolean *wasChanged);
```

Resolving and Reading Alias Records

```
pascal OSErr ResolveAlias   (const FSSpec *fromFile, AliasHandle alias,
                             FSSpec *target, Boolean *wasChanged);
pascal OSErr MatchAlias     (const FSSpec *fromFile,
                             unsigned long rulesMask,
                             const AliasHandle alias, short *aliasCount,
                             FSSpecPtr aliasList, Boolean *needsUpdate,
                             AliasFilterProcPtr aliasFilter,
                             Ptr yourDataPtr);
pascal OSErr GetAliasInfo   (const AliasHandle alias, AliasInfoType index,
                             Str63 theString);
```

Application-Defined Routine

```
pascal Boolean MyMatchAliasFilter
                            (CInfoPBPtr cpbPtr, Boolean *quitFlag,
                             Ptr myDataPtr);
```

Assembly-Language Summary

Data Structure

Alias Record Data Structure

0	`userType`	long	file type of target file
4	`aliasSize`	word	size, in bytes, of record

Trap Macros

Trap Macro Requiring Routine Selectors

_AliasDispatch

Selector	Routine
$0002	NewAlias
$0003	ResolveAlias
$0005	MatchAlias
$0006	UpdateAlias
$0007	GetAliasInfo
$0008	NewAliasMinimal
$0009	NewAliasMinimalFromFullPath
$000C	ResolveAliasFile

Result Codes

noErr	0	No error
nsvErr	–35	The volume is not mounted
fnfErr	–43	Target not found, but volume and parent directory found
paramErr	–50	Parameter error
dirNFErr	–120	Parent directory not found
usrCanceledErr	–128	The user canceled the operation

Disk Initialization Manager

Contents

This chapter describes the Disk Initialization Manager, the part of the Operating System that allows you to initialize disks and erase the contents of previously initialized disks. The Disk Initialization Manager provides a routine that allows you to present the standard user interface for initializing and naming disks. It also provides routines that allow you to initialize disks without presenting that standard user interface.

You need to read this chapter if your application does not mask out disk-inserted events. When your application receives a disk-inserted event, it must determine whether the inserted disk is valid. If the disk is not valid, your application can use the Disk Initialization Manager to present the user with the standard interface for initializing the disk.

To use this chapter, you should already be familiar with the Event Manager, which sends your application a disk-inserted event whenever a disk is inserted (unless you have masked out such events). You need to examine the `message` field of that event to determine whether the inserted disk is already initialized. You also need to be familiar with the File Manager if your application changes the default volume characteristics of newly initialized volumes.

This chapter begins by describing the operation of the Disk Initialization Manager, including

- formatting, verifying, and zeroing a disk
- the standard user interface for initializing and naming a disk
- bad block sparing

Then this chapter shows how you can

- determine whether an inserted disk is valid
- present the standard user interface to initialize and name an invalid disk
- present the standard user interface to erase a disk
- initialize or erase a disk without using the standard user interface
- change the default volume characteristics of newly initialized volumes

About the Disk Initialization Manager

The Disk Initialization Manager is the part of the Macintosh Operating System that manages the process of initializing disks. This package accepts requests to initialize a disk and translates them into control calls for the corresponding disk driver. The Disk Initialization Manager itself does not perform the low-level formatting or verification of the disk; instead, it simply manages the communication between the software requesting that a particular disk be initialized and the appropriate disk driver.

Note
In theory, you can use the Disk Initialization Manager to initialize any writable disk drive. In practice, however, most SCSI disk drivers ignore formatting control calls. Instead, low-level disk operations such as formatting and verification are usually performed by a utility program supplied with the disk. As a result, this chapter assumes that the disk to be initialized is a 3.5-inch floppy disk or an Apple Hard Disk 20SC, all of which are accessed through the Disk Driver. ◆

Usually, the Finder or the Standard File Package calls the Disk Initialization Manager when the user inserts an uninitialized disk. Occasionally the user will insert a disk when your application is frontmost. At that time, the Operating System generates a disk-inserted event. If your application has not masked out such events, it receives an event record for that event when it makes an event call and no events with higher priority are pending. You then need to determine whether the inserted disk is valid (as indicated by a value in the event record). If the disk is not valid, you should call the Disk Initialization Manager to allow the user to initialize the disk or, if desired, eject it.

If your application masks out disk-inserted events, the event stays in the event queue until your application calls the Standard File Package (which automatically processes disk-inserted events) or until the current application can handle disk-inserted events. In general, it's best not to mask out disk-inserted events and to handle them as described later in this chapter; otherwise, the user is likely to become confused when, after inserting an uninitialized or damaged disk, no disk icon appears on the desktop and no standard disk initialization dialog box appears. (Icons of initialized and undamaged disks always appear on the desktop, even if the current application ignores disk-inserted events.)

Disk Initialization

Disk initialization is the process of making a disk usable by the Macintosh Operating System. When shipped, most floppy disks are uninitialized because different operating systems have different initialization requirements. On Macintosh computers, disk initialization consists of three independent steps:

- disk formatting
- disk verification
- disk zeroing

All three steps must be performed successfully before the disk is considered initialized (or valid). You can use a single Disk Initialization Manager function, `DIBadMount`, to perform all three operations in sequence, or you can perform any one of them by calling a corresponding low-level function (either `DIFormat`, `DIVerify`, or `DIZero`). In general, your application should use the standard user interface described in the following section to initialize a disk.

The first step in the initialization process is **disk formatting.** Formatting a disk consists of writing special information onto a disk so that the disk driver can read from and write to the disk. This involves dividing the total usable space into sectors and tracks. See the

chapter “Disk Driver” in *Inside Macintosh: Devices* for a description of how a disk is divided into tracks and sectors.

The second step in the disk-initialization process is **disk verification.** Verifying a disk consists of reading every bit on the disk to ensure that the disk has been formatted correctly and contains no bad blocks. If an error occurs during the reading of any single bit, the verification is considered unsuccessful.

The third and final step in the disk-initialization process is **disk zeroing.** Zeroing a disk consists of creating on the disk the data structures and files necessary for the disk to be recognized as a hierarchical file system (HFS) volume. In particular, zeroing a disk places a master directory block (MDB), a volume bitmap, and a catalog file in appropriate locations on the disk. (For information on the locations and sizes of these items, see the description of the organization of data in a volume in the chapter “File Manager” in this book.) The volume bitmap and catalog file are set up to represent a volume containing no user files. As a result, zeroing a disk makes any files previously located on the disk inaccessible.

Beginning in system software version 7.0, zeroing a disk also causes the Disk Initialization Manager to attempt to remove any bad blocks (as identified during the disk-verification process) from the pool of available blocks on the disk. See “Bad Block Sparing” on page 5-7 for a description of this capability.

The Disk Initialization User Interface

The Finder and the Standard File Package both handle disk-inserted events for uninitialized disks by presenting a **disk initialization dialog box** asking the user whether the disk should be ejected or initialized. Your application too can easily call a Disk Initialization Manager routine that generates such a dialog box when the user inserts an invalid disk. Figure 5-1 illustrates one configuration of the dialog box.

Figure 5-1 The disk initialization dialog box

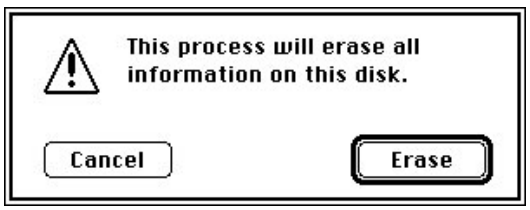
The appearance of the disk initialization dialog box changes to reflect changing conditions. For example, the icon changes to show which drive contains the disk. Also, the text of the dialog box changes according to what is wrong with the disk. The text might read “This is not a Macintosh disk” if the Disk Initialization Manager detects that the disk has been formatted for use on another operating system. Or, it might notify the user that a high-density disk can be used only on an Apple SuperDrive. Finally, if a user

inserts a single-sided disk into any disk drive, or a high-density disk into a high-density disk drive, then the Disk Initialization Manager changes the buttons in the dialog box, as illustrated in Figure 5-2, because such disks can be formatted in only one way.

Figure 5-2 Alternate buttons for the disk initialization dialog box

Regardless of the initial appearance of the disk initialization dialog box, it disappears if the user clicks Eject or Cancel. If, however, the user decides to initialize the disk, the text in the dialog box changes to warn the user that initialization erases any previous data on the disk, as illustrated in Figure 5-3.

Figure 5-3 The disk initialization warning



Finally, if the user decides to initialize the disk, the contents of the dialog box change so that the user can name the new disk, as illustrated in Figure 5-4.

Figure 5-4 The disk naming dialog box

After the user names the disk, the Disk Initialization Manager attempts to initialize it. If an error occurs and the initialization fails, an alert box notifies the user, and the disk is ejected.

The Disk Initialization Manager also provides a mechanism for using the standard interface to reinitialize disks that are already formatted. (This mechanism is useful, for example, when the user wants to reinitialize single-sided disks as double-sided disks.) The Finder takes advantage of this mechanism with its Erase Disk command, illustrated in Figure 5-5. After the user selects the erase operation from this dialog box, the reinitialization begins immediately, without further warnings. If desired, your application can use this same standard interface to allow users to reinitialize mounted disks (other than the startup volume). Your application can customize the text to be displayed in such a dialog box. Note that only a few utility applications actually need to provide users with this capability.

Figure 5-5 The Finder's disk erasing dialog box

If you are writing a utility program such as a disk-copying application, you might wish to initialize new disks or reinitialize valid disks without displaying the standard disk initialization dialog box. For example, your application might allow users to initialize multiple disks without having to respond to the standard dialog box each time. The Disk Initialization Manager provides low-level routines that allow you to do so. Unless you are writing a utility program of this type, you don't need to use these routines.

Bad Block Sparing

Beginning with system software version 7.0, the Disk Initialization Manager tries to initialize a disk even if it contains some bad blocks; this feature is called **bad block sparing.** Without bad block sparing, the Disk Initialization Manager considers a disk unusable even if just one block is bad. With bad block sparing, however, the Disk Initialization Manager attempts to work around the bad block by removing it from the pool of available free blocks. This prevents the File Manager from allocating the block to a file. Except in cases (described later) involving critical blocks on a disk, the Disk Initialization Manager can usually initialize a disk that would previously have been rejected as invalid. This section describes the operation of bad block sparing.

▲ **WARNING**
Applications that manipulate disks using File Manager routines are unaffected by bad block sparing. Software that accesses blocks directly from the disk or that makes assumptions about the physical blocks on a disk (such as a disk scavenger, recovery, or backup utility) is likely to fail or cause a loss of data on disks containing spared blocks. ▲

The bad block sparing occurs during the disk-zeroing phase of disk initialization. As a result, sparing occurs only when you call `DIZero` or `DIBadMount` (which internally calls `DIZero`), never when you call `DIFormat` or `DIVerify`. The only visible sign of the sparing process is an additional dialog box that contains the message "Re-Verifying Disk."

Disks without bad blocks are initialized exactly as in previous versions of system software. The sparing algorithm is invoked only if the disk verification fails during a call to the `DIBadMount` function or if the `DIZero` function encounters bad blocks during its zeroing. The sparing algorithm proceeds by making a second pass over the disk, writing and then reading back a test pattern. This testing is done a single track at a time. If any retries or errors occur during this test, all the sectors in the track are deemed bad.

If more than 25 percent of the disk is found to contain bad blocks, if the I/O errors appear to be due to hardware failure rather than media failure, or if certain critical sectors (described later) are bad, then the initialization fails just as it would have without the bad block sparing. Otherwise, the HFS volume structure is written to the disk. After the volume structure has been written, the Disk Initialization Manager performs several further operations during bad block sparing.

1. It sets the appropriate bits in the volume bitmap to indicate that the bad blocks are allocated to a file.
2. It creates file extent descriptors for the bad blocks and inserts them into the volume extents B*-tree so that the free-space scavenging that occurs at volume mount time (or that is done by disk utilities such as Disk First Aid) does not reintroduce the bad blocks into the volume bitmap. A special file ID (5) is used for these extents.
3. It sets bit 9 in the `drAtrb` field of the master directory block to indicate that bad blocks in the disk have been spared.
4. On 800K floppy disks only, it reduces the number of allocation blocks on the disk by 1 (from 1594 to 1593), to prevent previous versions of the Finder from doing disk-to-disk copies physically (that is, sector by sector). This copying operation would fail during an attempt to copy the bad blocks. The Finder does physical copies as an optimization only on disks containing exactly 1594 allocation blocks.

The critical sectors (those that must be good even on a spared disk) include the boot blocks, the master directory block and the spare master directory block, the volume bitmap, and the initial extents for the catalog and extents B*-tree files of the volume.

Notice that the bad block sparing algorithm does not create any new entries in the volume's catalog file. In other words, steps 1 and 2 of the algorithm trick the File Manager into thinking that the bad blocks have been allocated to some file, although no file is actually created to contain those blocks. For this reason, directory enumerations and file-by-file copies can proceed as they would have without bad block sparing. (If a file were created for the bad blocks, that file would need a parent directory; in that case, reading the catalog file to determine how many files that directory contains would produce erroneous results.)

Note
The bad block sparing capability described in this section applies only during disk initialization. The Operating System cannot correct problems that occur after a disk has been initialized (except by reinitializing the disk). ◆

Using the Disk Initialization Manager

The Disk Initialization Manager provides standard interfaces that allow your application

- to respond to the user's insertion of an unformatted or damaged disk by presenting the standard disk initialization dialog box
- to reinitialize valid disks, preserving their names but destroying their contents

You can override these standard interfaces by calling low-level Disk Initialization Manager routines, and you can also override the default volume characteristics that the Disk Initialization Manager gives to hierarchical volumes.

Responding to Disk-Inserted Events

When the user inserts a disk, the Operating System attempts to mount the volume on the disk by calling the File Manager function `PBMountVol`. If the volume is successfully mounted, an icon representing the disk appears on the desktop. The Operating System then generates a disk-inserted event. If the user is interacting with a standard file dialog box, the Standard File Package intercepts the disk-inserted event and handles it. Otherwise, the event is left in the event queue for your application to retrieve.

Your application must either mask out disk-inserted events or process them by checking to see whether the inserted disk is invalid. If you mask out such events, then each disk-inserted event needlessly occupies a position in the event queue until the user brings an application that can handle such events to the foreground or until your application invokes the Standard File Package. Also, displaying the disk initialization dialog box long after the disk has been inserted is likely to confuse the user. However, you might wish to mask out disk-inserted events when you create modal dialog boxes in which you process events with `WaitNextEvent` rather than `ModalDialog`. That way, your application can process disk-inserted events as soon as the modal dialog box closes.

Note
By default, the Dialog Manager's `ModalDialog` procedure automatically masks out disk-inserted events so that your application can handle them when dialog boxes close. If you wish to accept disk-inserted events in a modal dialog box in which you call `ModalDialog`, you must supply a filter procedure for the dialog box. See the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on how to write a filter procedure. ◆

Because handling disk-inserted events is easy, there is no good reason for your application to mask out the events in its main event loop. Listing 5-1 defines a procedure that your application can call when it receives a disk-inserted event.

Listing 5-1 Responding to disk-inserted events

```
PROCEDURE DoDiskEvent (myEvent: EventRecord);
VAR
   myPoint:    Point;
   myErr:      OSErr;
BEGIN
   IF HiWord(myEvent.message) <> noErr THEN
      BEGIN                              {attempt to mount was unsuccessful}
         DILoad;                         {load Disk Initialization Manager}
         SetPt(myPoint, 120, 120);       {set top left of dialog box}
         myErr := DIBadMount(myPoint, myEvent.message);
                                         {notify the user}
         DIUnload;                       {unload Disk Initialization Manager}
      END
   ELSE                                  {attempt to mount was successful}
      ;                                  {do other processing}
END;
```

The `DoDiskEvent` procedure in Listing 5-1 checks the high word of the event message to see if the disk is mounted properly. If it has not been mounted, `DoDiskEvent` calls the Disk Initialization Manager's `DIBadMount` function, which displays the disk initialization dialog box. Before doing so, `DoDiskEvent` calls `DILoad` to ensure that the Disk Initialization Manager and its dialog box are loaded into memory. If you did not call `DILoad` and the user started up with a floppy system startup disk, the Operating System might require that the user reinsert the system disk and might then attempt to initialize that disk. In Listing 5-1, if the user did start up with a floppy system startup disk on a single floppy-drive system, the `DILoad` procedure requests that the user insert the system disk so that it can read the necessary resources, and then it ejects that disk so that the user can again put the disk to be initialized into the drive. After calling `DIBadMount` to handle the uninitialized disk, `DoDiskEvent` calls `DIUnload` to release the resources `DILoad` read into memory.

Beginning with system software version 7.0, the first parameter to `DIBadMount` is ignored, and the disk initialization dialog box is automatically centered on the screen.

The procedure in Listing 5-1 ignores the result code returned by `DIBadMount` because ordinarily it does not concern your application. If an error does occur during initialization, `DIBadMount` informs the user and ejects the disk.

Erasing Initialized Disks

You can use the standard interface provided by the `DIBadMount` function to reinitialize disks that are already initialized correctly. Doing so permanently erases their contents, but does not change their names.

To reinitialize a disk, call `DIBadMount` with the high word of the event message equal to the result code `noErr`. The `DIBadMount` function presents the standard interface to initialize the disk in the drive whose number is specified by the low word of the event message. However, because the Disk Initialization Manager cannot know why your application wishes to reinitialize a disk, it cannot provide the initial text for the disk initialization dialog box. Therefore, your application must use the Dialog Manager's `ParamText` procedure to create a customized message, as illustrated in Listing 5-2.

If you need to reinitialize a valid disk but do not have access to the event message from when the disk was formatted, you can artificially create an event message by setting the event message to an integer representing the drive number, as follows:

```
myEvent.message := driveNum;
```

Doing so sets each of the high-order bits of the artificial event message to 0, which is desired because the constant `noErr` is equal to 0.

Listing 5-2 defines a procedure for displaying a disk initialization dialog box that allows the user to reinitialize the disk in the drive specified by `driveNum`. The disk initialization dialog box displays the text specified in the `myString` parameter. The procedure in Listing 5-2 in turn calls a procedure named `DoError`. You must define `DoError` to process the result code if the initialization did not successfully complete. The disk initialization dialog box does alert the user if the operation is not successfully completed, and the disk is ejected. However, your application might need to know that a formerly mounted disk is no longer mounted because reinitialization failed.

Listing 5-2 Reinitializing a valid disk

```
PROCEDURE DoEraseDisk (driveNum: Integer; myString: Str255);
VAR
   myPoint:          Point;
   myErr:            Integer;          {result code}
BEGIN
   DILoad;                             {load Disk Initialization Manager}
   ParamText(myString, '', '', '');    {set dialog text}
   SetPt(myPoint, 120, 120);           {set top left of dialog box}
   myErr := DIBadMount(myPoint, driveNum);
                                       {allow user to confirm erase}
   IF myErr <> noErr THEN
      DoError(myErr);                  {respond to error, if necessary}
   DIUnload;                           {unload Disk Initialization Manager}
END;
```

Overriding the Standard Initialization Interface

The disk initialization dialog box provides an easy-to-use, standard interface for initializing and reinitializing disks. However, if you wish, you can use three low-level Disk Initialization Manager functions that accomplish the three stages of disk initialization without presenting any user interface. The three functions are `DIFormat`, `DIVerify`, and `DIZero`. The `DIFormat` function attempts to format the disk, the `DIVerify` function verifies whether the format was successful, and the `DIZero` function updates the newly initialized volume's characteristics and attempts to spare any bad blocks on the disk.

Listing 5-3 shows how to reinitialize a disk without using the standard interface. The low-level functions work only if the disk is not already mounted in the disk drive. Therefore, Listing 5-3 uses high-level File Manager calls to unmount the volume and to remember the volume's name, so that it can be restored later. Because you are no longer using the standard interface, you must define the `DoError` procedure so that you can alert the user about an error.

Listing 5-3 Reinitializing a validly formatted disk without using the standard interface

```
PROCEDURE DoEraseDisk (driveNum: Integer);
VAR
   myErr:         OSErr;                    {result code}
   volName:       Str255;                   {name of volume}
   oldVRefNum:    Integer;                  {to unmount volume}
   oldFreeBytes:  LongInt;                  {for GetVInfo call}
BEGIN
   DILoad;                                  {load Disk Init. Manager}
   myErr := GetVInfo(driveNum, @volName, oldVRefNum, oldFreeBytes);
                                            {remember name of volume}
   IF myErr = noErr THEN
      myErr := UnmountVol(@volName, oldVRefNum);
                                            {unmount the disk}
   IF myErr = noErr THEN
      myErr := DIFormat(driveNum);          {format the disk}
   IF myErr = noErr THEN
      myErr := DIVerify(driveNum);          {verify format}
   IF myErr = noErr THEN
      myErr := DIZero(driveNum, volName);   {update volume information}
   IF myErr <> noErr THEN
      DoError(myErr);                       {respond to error}
   DIUnload;                                {unload Disk Init. Manager}
END;
```

If you wish, you can also respond to a user's insertion of an uninitialized or damaged disk by simply formatting the disk without using the standard interface. Listing 5-4 defines a procedure for this purpose. Listing 5-4 differs from Listing 5-3 only in that it does not begin by unmounting the volume (because the File Manager does not mount uninitialized or damaged disks).

Listing 5-4 Initializing an uninitialized disk without using the standard interface

```
PROCEDURE DoInitDisk (driveNum: Integer; volName: Str255);
VAR
   myErr:         OSErr;                    {result code}
BEGIN
   DILoad;                                  {load Disk Init. Manager}
   myErr := DIFormat(driveNum);             {format the disk}
   IF myErr = noErr THEN
      myErr := DIVerify(driveNum);          {verify format}
   IF myErr = noErr THEN
      myErr := DIZero(driveNum, volName);   {update volume information}
   IF myErr <> noErr THEN
      DoError(myErr);                       {respond to error}
   DIUnload;                                {unload Disk Init. Manager}
END;
```

Changing Default Volume Characteristics

The Disk Initialization Manager must set certain volume characteristics when it creates an HFS directory on a volume. Default values for these characteristics are stored in an HFS defaults record in ROM. If you wish, you can override those default values by placing a pointer to an HFS defaults record in the low-memory global variable `FmtDefaults`. The Disk Initialization Manager uses the record stored in ROM whenever this low-memory global variable contains `NIL`.

IMPORTANT

Most applications do not need to alter the default volume characteristics. This technique is useful primarily for applications, such as backup utilities, that intelligently adjust the allocation block size and clump size to maximize the amount of data written to a backup volume. ▲

The HFSDefaults data structure defines the HFS defaults record.

```
TYPE HFSDefaults =
RECORD
   sigWord:    PACKED ARRAY[0..1] OF Byte;   {signature word}
   abSize:     LongInt;                      {allocation block size in bytes}
   clpSize:    LongInt;                      {clump size in bytes}
   nxFreeFN:   LongInt;                      {next free file number}
   btClpSize:  LongInt;                      {B*-tree clump size in bytes}
   rsrv1:      Integer;                      {reserved}
   rsrv2:      Integer;                      {reserved}
   rsrv3:      Integer;                      {reserved}
END;
```

Field descriptions

`sigWord` The signature word to be used for newly initialized volumes. By default, this field is set to `'BD'` (hexadecimal $4244). You must set this field to `'BD'` for the volume to be recognized as an HFS volume.

`abSize` The number of bytes in each allocation block on newly initialized volumes. If you set this field to 0, the number of bytes in each allocation block is computed according to the following formula:

```
abSize = (1 + (blocks in volume/64K)) * 512 bytes
```

By default, this field is set to 0.

`clpSize` The number of bytes to be used for the clump on newly initialized volumes. By default, this field is set to `4*abSize`.

`nxFreeFN` The next free file number on newly initialized volumes. By default, this field is set to 16.

`btClpSize` The number of bytes to be used for the B*-tree clump on newly initialized volumes. If you set this field to 0, the number of bytes to be used for the B*-tree clump is computed according to the following formula:

```
btClpSize = ((blocks in volume)/128) * 512 bytes
```

By default, this field is set to 0.

`rsrv1` Reserved. Set to 0.

`rsrv2` Reserved. Set to 0.

`rsrv3` Reserved. Set to 0.

The code in Listing 5-5 fills in an `HFSDefaults` record, stores it in the system heap (so that the record remains in memory after the application terminates), and makes the low-memory global variable `FmtDefaults` a pointer to that record. Note that changing the default volume characteristics does not affect volumes that you have already initialized, but only volumes to be initialized.

Listing 5-5 Changing default volume characteristics

```
PROCEDURE ChangeHFSDefaults;
CONST
   FmtDefaults    = $039E;               {address of low-memory global}
TYPE
   HFSDefaultsPtr = ^HFSDefaults;        {pointer to override record}
   HFSDefaultsAdd = ^HFSDefaultsPtr;     {address of above pointer}
VAR
   myDefaults:    HFSDefaultsPtr;
BEGIN                                    {allocate record in system heap}
   myDefaults := HFSDefaultsPtr(NewPtrSysClear(SizeOf(HFSDefaults)));
   WITH myDefaults^ DO
      BEGIN
         ...                             {set fields of record}
      END;
   HFSDefaultsAdd(FmtDefaults)^ := myDefaults;
                                         {change value of global}
END;
```

If you later want to restore the default settings, you can reset the low-memory global variable `FmtDefaults` to `NIL`. Remember to delete any memory you have allocated.

Disk Initialization Manager Reference

This section describes the routines that are specific to the Disk Initialization Manager. See “Changing Default Volume Characteristics” on page 5-13 for a description of the Pascal data structure for the HFS defaults record.

Routines

The Disk Initialization Manager provides two routines (`DILoad` and `DIUnload`) that allow you to load and unload the package. The `DIBadMount` routine has two uses: to format uninitialized disks that the user inserts and to reinitialize volumes by erasing their data without changing their names. Last, three low-level routines (`DIFormat`, `DIVerify`, and `DIZero`) allow you to perform the steps of formatting, verifying, and zeroing the disk separately.

Loading and Unloading the Disk Initialization Manager

Even a user with a hard disk drive might occasionally use a floppy disk to start up the computer. When you call the Disk Initialization Manager to initialize a disk, it might need to read a resource from the System resource file. If the disk containing the System

resource file is not already mounted, the user might need to switch disks, and system software might accidentally try to reinitialize the startup volume. The `DILoad` procedure allows you to avoid this problem by ensuring that the resources the Disk Initialization Manager needs are preloaded into memory. The `DIUnload` procedure reverses the effects of `DILoad`.

DILoad

You can use the `DILoad` procedure to ensure that the Disk Initialization Manager and its associated dialog box and dialog items are in memory.

```
PROCEDURE DILoad;
```

DESCRIPTION

The `DILoad` procedure reads the Disk Initialization Manager and its associated dialog box and dialog items into memory and makes them unpurgeable. Depending on which Macintosh model the user is using, the Disk Initialization Manager and the dialog box and dialog items are either in ROM or in the System file.

Ordinarily, you call the `DILoad` procedure when you anticipate that the user will need to format a disk. The Standard File Package automatically calls `DILoad` when you call `StandardGetFile` or `StandardPutFile`. If you are writing a utility program that frequently needs to initialize disks, such as a disk-copying program, you might call `DILoad` at the beginning of your application.

When you use the low-level disk-initialization routines `DIFormat`, `DIVerify`, and `DIZero`, the Disk Initialization Manager does not need to load a dialog box. Therefore, if you use only these routines, you can (if you wish) call the Resource Manager to read just the package resource into memory and the Memory Manager procedure to make it unpurgeable. To read just the package resource into memory, you can call the `GetResource` function with a resource ID of 2 and a resource type of `'PACK'`. Then, you need to use the `HNoPurge` procedure to make the package resource unpurgeable.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DILoad` are

Trap macro	Selector
`_Pack2`	$0002

SPECIAL CONSIDERATIONS

Because the `DILoad` procedure allocates memory, you should not call it at interrupt time.

DIUnload

To free the memory space occupied by the Disk Initialization Manager, you can call the `DIUnload` procedure.

```
PROCEDURE DIUnload;
```

DESCRIPTION

The `DIUnload` procedure makes the Disk Initialization Manager and its associated dialog box and dialog items purgeable. They remain in memory until the Memory Manager purges the heap zone.

If you are using the low-level disk initialization routines and read just the package resource into memory, you can free the memory the package occupies by calling the `ReleaseResource` procedure.

To force the Memory Manager to purge the heap zone so that it really frees the memory occupied by the Disk Initialization Manager and its dialog box and dialog items, you can call one of the Memory Manager routines `PurgeMem` and `MaxMem`. For more information, see the chapter "Memory Manager" in *Inside Macintosh: Memory*.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DIUnload` are

Trap macro	Selector
`_Pack2`	$0004

SPECIAL CONSIDERATIONS

Because `DIUnload` might affect memory, you should not call it at interrupt time.

Initializing a Disk

You can use the Disk Initialization Manager to initialize uninitialized disks and to reinitialize previously initialized disks. The `DIBadMount` function accomplishes both tasks.

DIBadMount

To respond to the user's insertion of an uninitialized or damaged disk, you can call the `DIBadMount` function.

```
FUNCTION DIBadMount (where: Point; evtMessage: LongInt): Integer;
```

`where` The desired location, in global coordinates, of the upper-left corner of the disk initialization dialog box. In system software versions 7.0 and later, this parameter is ignored, and the dialog box is automatically centered on the screen.

`evtMessage` The event message received when the disk is inserted. The high word of this message contains the result code associated with the disk insertion. The low word of this message indicates the number of the drive into which the user inserted the disk.

DESCRIPTION

The `DIBadMount` function evaluates the result code in the high word of the `evtMessage` parameter and responds appropriately. If the result code is `noErr`, the function allows the user to erase the contents of the disk. If the result code is `ioErr`, `badMDBErr`, or `noMacDskErr`, initializing the disk might correct the problem, and so `DIBadMount` displays a dialog box that explains the problem and allows the user to initialize the disk. If the result code is `extFSErr`, `memFullErr`, `nsDrvErr`, `paramErr`, or `volOnLinErr`, then initializing the disk would not correct the problem. In this case, `DIBadMount` ejects the disk from the drive and returns the result code.

Before presenting the disk initialization dialog box, `DIBadMount` checks whether the drive contains an already mounted volume. If so, it ejects the disk and returns 2 as its result. This happens rarely and could reflect an error in your application (for example, you forgot to call `DILoad`, and the user had to switch to the disk containing the System resource file).

The `DIBadMount` function uses just one disk initialization dialog box to cover all disk initialization situations. The dialog box contains many dialog items, which are hidden and shown as appropriate. The dialog box always contains an icon indicating the drive containing the disk to be initialized.

The initial text of the disk initialization dialog box depends on the result code received. For example, if you pass `noMacDskErr` to `DIBadMount` in the `evtMessage` parameter, the dialog box displays the text "This is not a Macintosh disk." If you pass the result code `noErr`, you can customize the message by using the Dialog Manager's `ParamText` procedure.

The disk initialization dialog box contains a button allowing the user to cancel the initialization and one or two buttons allowing the user to request initialization of the disk. Usually, the cancel button is labeled Eject, but if the result code passed to `DIBadMount` within the `evtMessage` parameter is `noErr`, then the cancel button is labeled Cancel. If the user responds to the disk initialization dialog box by clicking the Eject button, `DIBadMount` ejects the disk and returns 1 as its result. If the user clicks the Cancel button, `DIBadMount` returns 1 but does not eject the disk.

In most cases, the Initialize button is the only alternative to the Eject or Cancel button. However, if the user inserts a double-sided (but not high-density) disk into a double-sided or high-density disk drive, `DIBadMount` presents buttons labeled One-Sided and Two-Sided. The user can then decide whether to make the disk single-sided or double-sided. If the user clicks the Initialize button, the One-Sided button, or the Two-Sided button, `DIBadMount` warns the user that the initialization process erases any existing data on the disks. If the user proceeds, `DIBadMount` allows the user to name the disk if it is not already named and then updates the text of the dialog box to inform the user of the progress of the operation. If the operation fails, `DIBadMount` alerts the user and ejects the disk, returning an appropriate result code.

You can use `DIBadMount` to format hard disks as well as floppy disks. However, you should not attempt to format the startup volume.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DIBadMount` are

Trap macro	Selector
`_Pack2`	$0000

SPECIAL CONSIDERATIONS

Because the `DIBadMount` function might allocate memory, you should not call it at interrupt time.

RESULT CODES

`[no name]`	2	Disk in specified drive is already mounted
`[no name]`	1	User canceled initializing
`noErr`	0	No error
`paramErr`	–50	Drive number specified is bad
`volOnLinErr`	–55	Volume is already online
`nsDrvErr`	–56	No such drive
`extFSErr`	–58	Disk has external file system
`lastDskErr`	–64	Last of the range of low-level disk errors
`...`		
`firstDskErr`	–84	First of the range of low-level disk errors
`memFullErr`	–108	Not enough memory

Low-Level Disk Initialization Routines

If you do not want to use the standard interface for initializing uninitialized volumes, you can use the Disk Initialization Manager's low-level routines. For example, if you are writing a disk-copying application, initializing a disk might be only part of the copying process. In this case, you might wish to create your own dialog boxes warning the user about the repercussions of initializing a disk and giving information on the progress of the initialization.

The three low-level disk-initialization routines are `DIFormat`, `DIVerify`, and `DIZero`. Ordinarily, you call them in that order to format an uninitialized disk, to verify the format, and to set the volume's volume information block and catalog.

DIFormat

To format a disk, you can use the `DIFormat` function.

```
FUNCTION DIFormat (drvNum: Integer): OSErr;
```

`drvNum` The number of the drive containing the disk to be formatted.

DESCRIPTION

The `DIFormat` function attempts to format the disk in the drive specified by the `drvNum` parameter and returns a result code indicating whether it completed the formatting successfully or failed. Formatting a disk consists of writing special information onto it so that the disk driver can read from and write to the disk.

You can use `DIFormat` to format any unlocked disk, including single-sided disks, double-sided disks, high-density disks, and hard disk drives. It formats both sides of a double-sided disk.

You have to unmount a disk before calling the `DIFormat` function.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DIFormat` are

Trap macro	Selector
`_Pack2`	$0006

SPECIAL CONSIDERATIONS

You should not call `DIFormat` at interrupt time.

RESULT CODES

`noErr`	0	No error
`volOnLinErr`	–55	Volume is online
`lastDskErr`	–64	Last of the range of low-level disk errors
`...`		
`firstDskErr`	–84	First of the range of low-level disk errors

DIVerify

To verify a disk you have formatted, you can use the `DIVerify` function.

```
FUNCTION DIVerify (drvNum: Integer): OSErr;
```

`drvNum` The number of the drive containing the disk to be verified.

DESCRIPTION

The `DIVerify` function verifies the format of the disk in the drive specified by the `drvNum` parameter. It reads each bit from the disk and returns a result code indicating whether all bits were read successfully or not. The `DIVerify` function does not affect the contents of the disk itself.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DIVerify` are

Trap macro	Selector
`_Pack2`	$0008

SPECIAL CONSIDERATIONS

You should not call `DIVerify` at interrupt time.

RESULT CODES

`noErr`	0	No error
`lastDskErr`	–64	Last of the range of low-level disk errors
`...`		
`firstDskErr`	–84	First of the range of low-level disk errors

DIZero

To complete the disk-initialization process, you can use the `DIZero` function.

```
FUNCTION DIZero (drvNum: Integer; volName: Str255): OSErr;
```

`drvNum` The number of the drive containing the disk to be zeroed.

`volName` The name of the volume (to be included in the volume information).

DESCRIPTION

On the unmounted volume in the drive specified by the given drive number, the `DIZero` function sets the volume information, the volume bitmap, a file directory, and the desktop database (or desktop file) to the settings corresponding to a volume with no

files. This function completes the process of making any files previously on the volume permanently inaccessible. If the operation fails, `DIZero` returns a result code indicating that a low-level disk error occurred. Otherwise, it mounts the volume by calling the File Manager function `PBMountVol` and returns that function's result code.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `DIZero` are

Trap macro	Selector
`_Pack2`	$000A

SPECIAL CONSIDERATIONS

You should not call `DIZero` at interrupt time. In system software version 7.0 and later, `DIZero` automatically performs bad block sparing, as described in "Bad Block Sparing," beginning on page 5-7.

RESULT CODES

`noErr`	0	No error
`ioErr`	–36	I/O error
`paramErr`	–50	Drive number specified is bad
`volOnLinErr`	–55	Volume is already online
`nsDrvErr`	–56	No such drive
`noMacDskErr`	–57	Disk is not a Macintosh disk
`extFSErr`	–58	Disk has external file system
`badMDBErr`	–60	Master directory block is bad
`lastDskErr`	–64	Last of the range of low-level disk errors
`...`		
`firstDskErr`	–84	First of the range of low-level disk errors
`memFullErr`	–108	Not enough memory

Summary of the Disk Initialization Manager

Pascal Summary

Data Types

HFS Defaults Record

```
TYPE HFSDefaults =
RECORD
   sigWord:    PACKED ARRAY[0..1] OF Byte;   {signature word}
   abSize:     LongInt;                      {allocation block size in bytes}
   clpSize:    LongInt;                      {clump size in bytes}
   nxFreeFN:   LongInt;                      {next free file number}
   btClpSize:  LongInt;                      {B*-tree clump size in bytes}
   rsrv1:      Integer;                      {reserved}
   rsrv2:      Integer;                      {reserved}
   rsrv3:      Integer;                      {reserved}
END;
```

Routines

Loading and Unloading the Disk Initialization Manager

```
PROCEDURE DILoad;
PROCEDURE DIUnload;
```

Initializing a Disk

```
FUNCTION DIBadMount         (where: Point; evtMessage: LongInt): Integer;
```

Low-Level Disk-Initialization Routines

```
FUNCTION DIFormat           (drvNum: Integer): OSErr;
FUNCTION DIVerify           (drvNum: Integer): OSErr;
FUNCTION DIZero             (drvNum: Integer; volName: Str255): OSErr;
```

C Summary

Data Types

HFS Defaults Record

```
struct HFSDefaults {
      char        sigWord[2];    /*signature word*/
      long        abSize;        /*allocation block size in bytes*/
      long        clpSize;       /*clump size in bytes*/
      long        nxFreeFN;      /*next free file number*/
      long        btClpSize;     /*B-Tree clump size in bytes*/
      short       rsrv1;         /*reserved*/
      short       rsrv2;         /*reserved*/
      short       rsrv3;         /*reserved*/
};

typedef struct HFSDefaults HFSDefaults;
```

Routines

Loading and Unloading the Disk Initialization Package

```
pascal void DILoad          (void);
pascal void DIUnload        (void);
```

Initializing a Disk

```
pascal short DIBadMount     (Point where, long evtMessage);
```

Low-Level Disk-Initialization Routines

```
pascal OSErr DIFormat       (short drvNum);
pascal OSErr DIVerify       (short drvNum);
pascal OSErr DIZero         (short drvNum, const Str255 volName);
```

Assembly-Language Summary

Data Structures

HFSDefaults Data Structure

0	sigWord	word	signature word
2	abSize	long	allocation block size in bytes
6	clpSize	long	clump size in bytes
10	nxFreeFN	long	next free file number
14	btClpSize	long	B*-tree clump size in bytes
18	rsrv1	word	reserved
20	rsrv2	word	reserved
22	rsrv3	word	reserved

Trap Macros

Trap Macro Requiring Routine Selectors

_Pack2

Selector	Routine
$0000	DIBadMount
$0002	DILoad
$0004	DIUnload
$0006	DIFormat
$0008	DIVerify
$000A	DIZero

Global Variables

FmtDefaults	long	Pointer to substitute values for hierarchical volume directories.

Result Codes

[no name]	2	Disk in specified drive is already mounted
[no name]	1	User canceled initializing
noErr	0	No error
ioErr	–36	I/O error
paramErr	–50	Drive number specified is bad
volOnLinErr	–55	Volume is already online
nsDrvErr	–56	No such drive

`noMacDskErr`	–57	Disk is not a Macintosh disk
`extFSErr`	–58	Disk has external file system
`badMDBErr`	–60	Master directory block is bad
`lastDskErr`	–64	Last of the range of low-level disk errors
`firstDskErr`	–84	First of the range of low-level disk errors
`memFullErr`	–108	Not enough memory

Glossary

absolute search A search that begins at the root directory of the file system hierarchy and always descends the hierarchy. See also **relative search.**

access modes A set of file permissions that specify what abilities should be allowed to a user attempting to open a file fork. See also **deny modes.**

access path A description of the route that the File Manager follows to access a file; created when a file is opened. See also **file reference number.**

access permissions See **access modes, file permissions.**

access privileges See **directory access privileges.**

access rights The permissions governing the access to a file, or the privileges governing the access to a directory.

activation procedure An application-defined procedure that controls the highlighting of application-defined dialog items capable of receiving keyboard input.

active field The target of keyboard input in a dialog box.

AFP volume A volume that is accessed using the AppleTalk Filing Protocol.

alias An object in the file system that represents another file, directory, or volume.

Alias Manager The part of the Operating System that helps you to locate specified files, directories, or volumes at a later time. The Alias Manager creates and resolves alias records.

alias record A data structure created by the Alias Manager to identify a file, directory, or volume.

alias target The file, directory, or volume described by an alias record.

allocation block A group of consecutive logical blocks on a volume.

AppleTalk Filing Protocol (AFP) A protocol that allows users to share data files and application programs that reside in a shared resource, such as a file server.

asynchronous execution A mode of invoking a routine. During the asynchronous execution of a routine, an application is free to perform other tasks.

backing-store file The file that the Virtual Memory Manager uses to store the contents of unneeded pages of memory.

bad block sparing The process of working around a bad block by removing it from the pool of available free blocks.

blank access privileges The directory access privileges under which a directory has the same access privileges as the directory's parent.

block A group regarded as a unit; usually refers to data or memory in which data is stored. See also **allocation block.**

boot blocks The blocks on a disk that contain system startup information.

browsing access The file access permissions that allow users to read but not modify a file.

B*-tree A method of organizing information into a collection of nodes. The nodes are arranged in a way that allows efficient access to the stored information.

B*-tree control block A block of memory that contains information about a B*-tree file (either a catalog file or an extents overflow file).

B*-tree file A file that is organized as a B*-tree. See also **catalog file, extents overflow file.**

B*-tree header record A record in a header node that contains information about the beginning of the tree, as well as the size of the tree.

catalog file A special file, located on a volume, that contains information about the hierarchical organization of files and folders on that volume.

catalog node An entry in a volume's catalog file that describes either a file or a directory.

catalog node ID A unique number assigned to a node in a catalog file. For a directory, the catalog node ID is the directory ID; for a file, the catalog node ID is the file ID.

closed file A file without an access path. You cannot read from or write to closed files.

clump A group of contiguous allocation blocks. Space is allocated to a new file in clumps to promote file contiguity and avoid fragmentation.

clump size The number of allocation blocks to be allocated to a new file.

CNID See **catalog node ID.**

CNode See **catalog node.**

common parent The lowest-level directory that appears in the pathnames of two objects on a volume.

completion routine A routine that is executed when an asynchronous call to some other routine is completed.

current directory The directory whose contents are listed in the dialog box displayed by the Standard File Package. See also **default directory.**

current disk The current volume.

current volume The volume on which the current directory is located.

data buffer A buffer (usually in an application's heap) that contains information to be written to a file from the application, or read from a file to an application.

data fork The part of a file that contains data accessed using the File Manager.

default directory The directory used in File Manager routines whenever you don't explicitly specify some directory. See also **current directory.**

default volume The volume that contains the default directory.

deny modes A set of file permissions that specify what abilities should be denied to users attempting to open a file fork already opened by another user. See also **access modes.**

dialog hook function An application-defined function that handles item selections in a dialog box displayed by the Standard File Package.

directory A subdivision of a volume, available in the hierarchical file system. A directory can contain files and other directories (known as subdirectories).

directory access privileges A set of conventions for controlling access to a directory.

directory ID A unique number assigned to a directory. The File Manager uses this number to distinguish a directory from others on the same volume. See also **catalog node ID.**

disk A physical medium capable of storing information.

disk cache A part of RAM that acts as an intermediate buffer when data is read from and written to file systems on secondary storage devices.

disk formatting The process of writing special information onto a disk so that the disk driver can read from and write to the disk.

disk initialization The process of making a disk usable by the Macintosh Operating System.

disk initialization dialog box A dialog box asking the user whether a disk should be ejected or initialized.

Disk Initialization Manager The part of the Macintosh Operating System that manages the process of initializing disks.

disk-inserted event An event generated when the user inserts a disk in a disk drive or takes any other action that requires a volume to be mounted.

disk switch dialog box A dialog box asking the user to insert a particular disk.

disk verification The process of reading every bit on the disk to ensure that the disk has been formatted correctly and contains no bad blocks.

disk zeroing The process of creating on the disk the data structures and files necessary for the disk to be recognized as a hierarchical file system (HFS) volume.

display list In a standard file dialog box, the list of files, folders, and volumes at one level of the display hierarchy, from which the user can select items.

document A file that a user can create and edit. A document is usually associated with a single application, which the user expects to be able to open by double-clicking the document's icon in the Finder.

document record An application-defined data structure that contains information about the window, any controls in the window (such as scroll bars), and the file (if any) whose contents are displayed in the window.

drive queue A list of all volumes connected to the computer.

end-of-file (EOF) See **logical end-of-file, physical end-of-file.**

EOF See **logical end-of-file, physical end-of-file.**

exclusive access The file access permissions that deny other users both read and write access to a file.

exhaustive search A search using an algorithm that scans an entire volume to look for possible matches.

extent A contiguous range of allocation blocks that have been allocated to some file.

extent data record A data record that contains three extent descriptors. Extent data records are stored in the leaf nodes of the extents overflow file, in the catalog file, and in the boot blocks.

extent descriptor A description of an extent, consisting of the number of the first allocation block of the extent followed by the length of the extent. Defined by the `ExtDescriptor` data type.

extents overflow file A special file containing all extent data records that are not stored elsewhere by the File Manager.

fast search A search that employs an algorithm designed to find the target of an alias record quickly. See also **absolute search.**

FCB See **file control block.**

file A named, ordered sequence of bytes stored on a Macintosh volume. A file is divided into a data fork and a resource fork.

file access permissions See **file permissions.**

file control block (FCB) A fixed-length data structure, contained in the file-control-block buffer, where information about an access path to a file is stored.

file-control-block buffer A block in the system heap that contains one file control block for each access path.

file filter function An application-defined function that helps determine which files appear in the list of files to open. This list appears in the dialog boxes displayed by the Standard File Package.

file fork One of the two parts of a file. See also **data fork, resource fork.**

file ID A unique number assigned to a file. The File Manager uses this number to distinguish a file from others on the same volume. See also **catalog node ID.**

file ID reference An internal record in the volume's catalog file. This record specifies the filename and parent directory ID of the file with a given file ID**.**

file ID thread record See **file ID reference.**

file I/O queue A queue containing parameter blocks for all I/O requests to the File Manager.

File Manager The part of the Macintosh Operating System that manages the organization, reading, and writing of data located on physical data storage devices such as disk drives.

file mark A marker the File Manager uses to keep track of its place in a file during a read or write operation. The file mark specifies the position of the next byte that will be read or written.

filename A sequence of up to 31 printing characters, excluding colons, that identifies a file.

file permissions A set of conventions for controlling access to a file. A file's permissions consist of **access modes** and **deny modes.**

file reference number A number (greater than 0) that is returned to your application when it opens a fork of a file using File Manager routines; each file reference number corresponds to a unique access path.

file server A computer running software that provides network users with access to shared disks or other mass-storage devices.

file system A method of organizing files and directories on a volume.

file system specification A record that identifies a stored file or directory by volume reference number, parent directory ID, and name. Defined by the FSSpec data type.

Finder A Macintosh application that allows access to documents and other applications. The Finder uses icons to represent objects on a volume.

flush To write data from a cache in memory to a volume.

folder A directory. See **directory.**

fork See **file fork.**

formatting See **disk formatting.**

full pathname A pathname that begins in the root directory.

guest A user who is logged on to a file server without a registered user name and password.

header node The first node in a B*-tree file; it contains essential information about the entire B*-tree file.

HFS See **hierarchical file system.**

HFS volume A volume that is organized according to the hierarchical file system.

hierarchical file system (HFS) A method of organizing files and directories on a volume in a hierarchical or tree-like structure.

index node A node containing records that point to other nodes in the B*-tree hierarchy.

initialization See **disk initialization.**

I/O queue See **file I/O queue.**

I/O request A request for input from or output to a file or device driver; caused by calling a File Manager or Device Manager routine asynchronously.

leaf node A node that contains data records.

locked file A file whose data cannot be changed.

locked range A range of bytes in a file whose data cannot be changed.

locked volume A volume whose data cannot be changed.

logical block A portion of a volume. Usually 512 bytes long.

logical end-of-file The position of 1 byte past the last byte in a file; equal to the actual number of bytes in the file.

log on To connect to a networked file server or to a local machine that requires user authentication. Usually a user must specify a user name and password to be able to log on to a file server.

Macintosh file system (MFS) A now-obsolete method of organizing files on a volume in a "flat" or nonhierarchical structure. See also **hierarchical file system.**

Make Changes privileges The directory access privileges that allow other users to create, rename, delete, and write files in the specified directory.

map node A node that contains an additional map record.

map record A record in a header node or map node that indicates which nodes in a B*-tree file are used and which are not**.**

mark See **file mark.**

master directory block (MDB) The part of a volume that contains information about the volume, such as the volume name and allocation block size.

MFS volume A volume that is organized using the Macintosh file system.

modal-dialog filter function An application-defined function that filters events passed from the Event Manager to the Standard File Package when one of its dialog boxes is being displayed.

modes See **access modes, deny modes.**

mount To make a volume available on the local machine.

mounted volume A volume that has had its descriptive information read by the File Manager and placed into a volume control block in memory.

newline character Any character, but usually the Return character (ASCII code $0D), that indicates the end of a sequence of bytes.

newline mode A mode of reading data in which the end of the data is indicated by a newline character (and not by a specific byte count).

node A part of a B*-tree.

node descriptor The first part of a B*-tree node; it contains information about the node, as well as forward and backward links to other nodes**.**

offline volume A volume that has been mounted but made temporarily unavailable (for example, because it was ejected).

offspring For a given directory, the set of files and directories the given directory contains.

online volume A volume that has been mounted and is currently available for File Manager operations.

open file A file with an access path. You can read from and write to open files only.

open permission Information about a file that indicates whether the file can be read from, written to, or both.

parent directory The directory in which a file or directory is located.

parent directory ID The directory ID of the directory containing a file or directory.

partial pathname A pathname that begins in some directory other than the root directory.

partition A part of a disk that has been allocated to a particular operating system, file system, or device driver.

partition map A block of information that describes the organization of partitions on a disk.

password A string of characters that a user or application must provide to gain access to a networked file server or to a local machine that requires user authentication. Passwords are frequently encrypted prior to transmission over a network to ensure network security.

pathname A series of concatenated directory names and filenames that identifies a given file or directory. See also **full pathname, partial pathname.**

path reference number See **file reference number.**

permissions See **file permissions.**

physical end-of-file The position of 1 byte past the last allocation block of a file; equal to 1 more than the maximum number of bytes the file can contain.

pointer record The kind of record contained in an index node in a B*-tree file. The structure of a pointer record depends on the kind of B*-tree in which it is contained.

poor man's search path The list of directories that the File Manager searches whenever it cannot find a specified file in the specified directory.

preferences file A file that stores a user's settings for a document or application.

Preferences folder A directory located in the System Folder that stores preferences files.

privilege model A set of conventions for controlling access to stored files and directories.

privileges See **directory access privileges.**

pseudo-item A constant that does not represent any actual item in the dialog list of one of the dialog boxes displayed by the Standard File Package.

range locking Locking a range of bytes in a file so that other users can't read from or write to that range, but allowing the rest of the file to be accessed.

read privileges See **See Files privileges.**

read/write permission Information associated with an access path that indicates whether the file can be read from, written to, or both.

relative path A path to the target from another file or directory on the same volume.

relative search A search that starts in a specified directory and searches for the target of an alias record by ascending the file system hierarchy to a predetermined common parent of the target and the starting directory, and then descending the hierarchy from that common parent.

resolve To find the target of an alias record.

resource fork The fork of a file that contains the file's resources.

root directory The directory at the base of a volume.

root node The first index node in a B*-tree.

search key A piece of data that the File Manager uses when searching through a B*-tree to locate the information it needs**.**

search privileges See **See Folders privileges.**

See Files privileges The directory access privileges that allow users to read files in the specified directory.

See Folders privileges The directory access privileges that allow users to see other directories in the specified directory.

shared access The file access permissions that allow other users both read and write access to a file.

shared environment Any operating environment that supports multiple users and multiple access to data or applications.

share point A volume or directory made available for sharing on the network.

single-writer access The file access permissions that deny other users write access to a file but allow them to read it.

Standard File Package The part of system software that allows you to present the standard user interface when a file is to be saved or opened.

subdirectory A directory that is contained in some other directory. All directories on a volume except the root directory are subdirectories.

synchronous execution A mode of invoking a routine. After calling a routine synchronously, an application cannot perform other tasks until the routine is completed.

system startup information Certain configurable system parameters that are stored in the boot blocks of a volume and read in at system startup.

target See **alias target.**

unmounted volume A volume that hasn't yet been mounted, or a volume that was previously mounted but has since had its volume control block removed from the VCB queue.

user authentication method A process used by a file server or workstation to confirm the user's identity.

user name A string of characters that uniquely identifies a user for login purposes.

VCB See **volume control block.**

VCB queue See **volume control block queue.**

verification See **disk verification.**

volume A portion of a storage device that is formatted to contain files.

volume bitmap A data structure that contains a series of bits indicating which blocks on the volume are allocated. Volume bitmaps exist both on HFS volumes and in memory.

volume catalog See **catalog file.**

volume control block (VCB) A nonrelocatable block of memory in the system heap that contains information about a specific mounted volume, including the information from the volume's master directory block.

volume control block queue A list of the volume control blocks for all mounted volumes.

volume index A number identifying the position of a mounted volume listed in the volume control block queue.

volume information block (VIB) See **master directory block.**

volume name A sequence of up to 27 characters, excluding colons (:), that identifies a volume.

volume reference number A unique number assigned to a volume when it's mounted; used to refer to the volume.

working directory A temporary directory reference by which the File Manager specifies both a directory and the volume on which it resides. The File Manager assigns a reference number to each working directory.

working directory control block A data structure that contains the directory ID of a working directory as well as the volume reference number of the volume on which the directory is located.

working directory reference number A temporary reference number that encodes a directory ID and a volume reference number. It can be used in place of the volume reference number in most File Manager calls.

write privileges See **Make Changes privileges.**

zeroing See **disk zeroing.**

Index

A

B

C

D

E

F

G

H

I, J

K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y

Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter IINTX printer. Final page negatives were output directly from text files on an AGFA ProSet 9800 imagesetter. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITER
Tim Monroe

DEVELOPMENTAL EDITOR
Antonio Padial

ILLUSTRATOR
Peggy Kunz

PRODUCTION EDITOR
Rex Wolf

PROJECT MANAGER
Patricia Eastman

COVER DESIGNER
Barbara Smyth

Special thanks to Lars Borresen, Bill Bruffey, Dave Feldman, Nick Kledzik, Prashant Patel, and Kenny Tung.

Acknowledgments to Michael Abramowicz, Neil Day, Richard Dizmang, Sharon Everson, Sanborn Hodgkins, Jim Luther, Sue Luttner, Mikey McDougall, Jim Reekes, Keith Rollin, Gordon Sheridan, Steve Szymanski, and the entire *Inside Macintosh* team.