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.

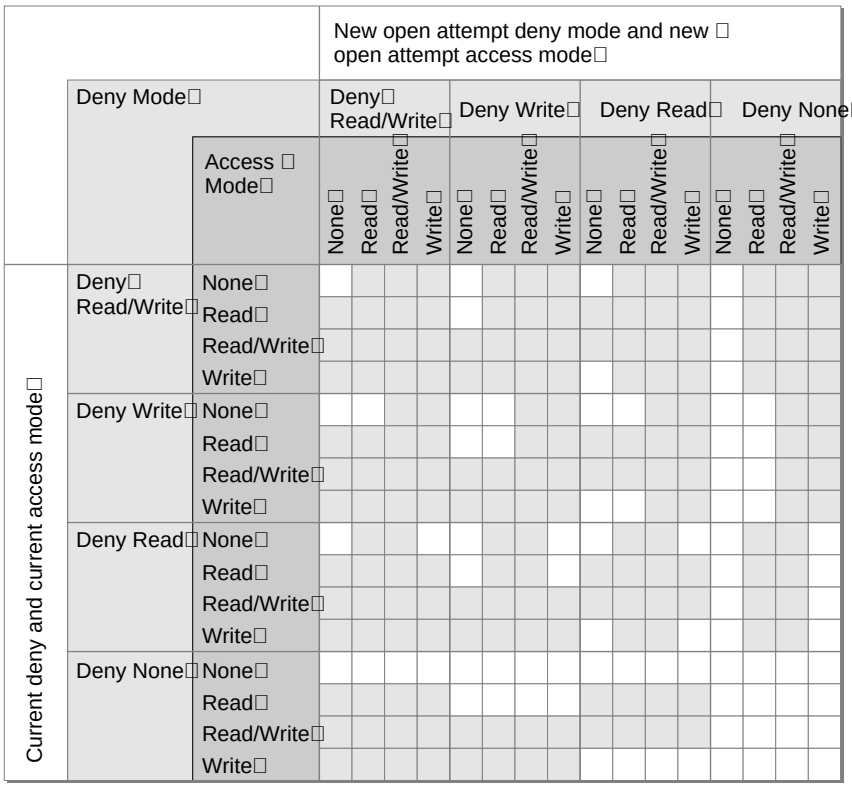
Shared File Access Permissions

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

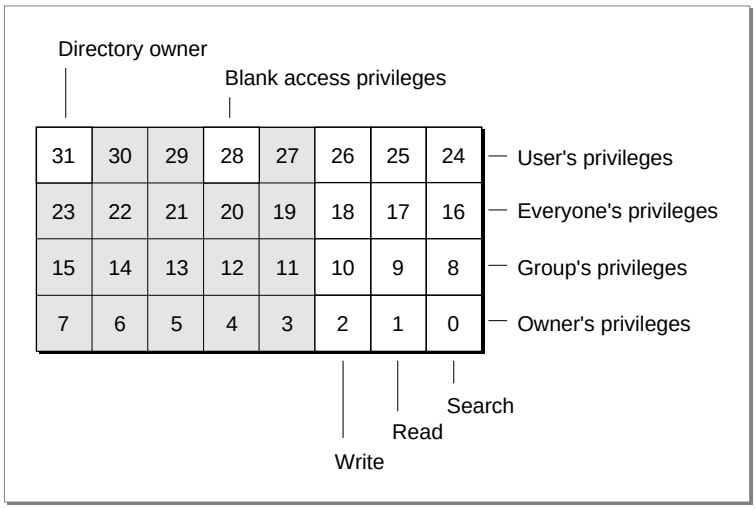
You cannot assign blank access privileges to a volume's root directory. ◆

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

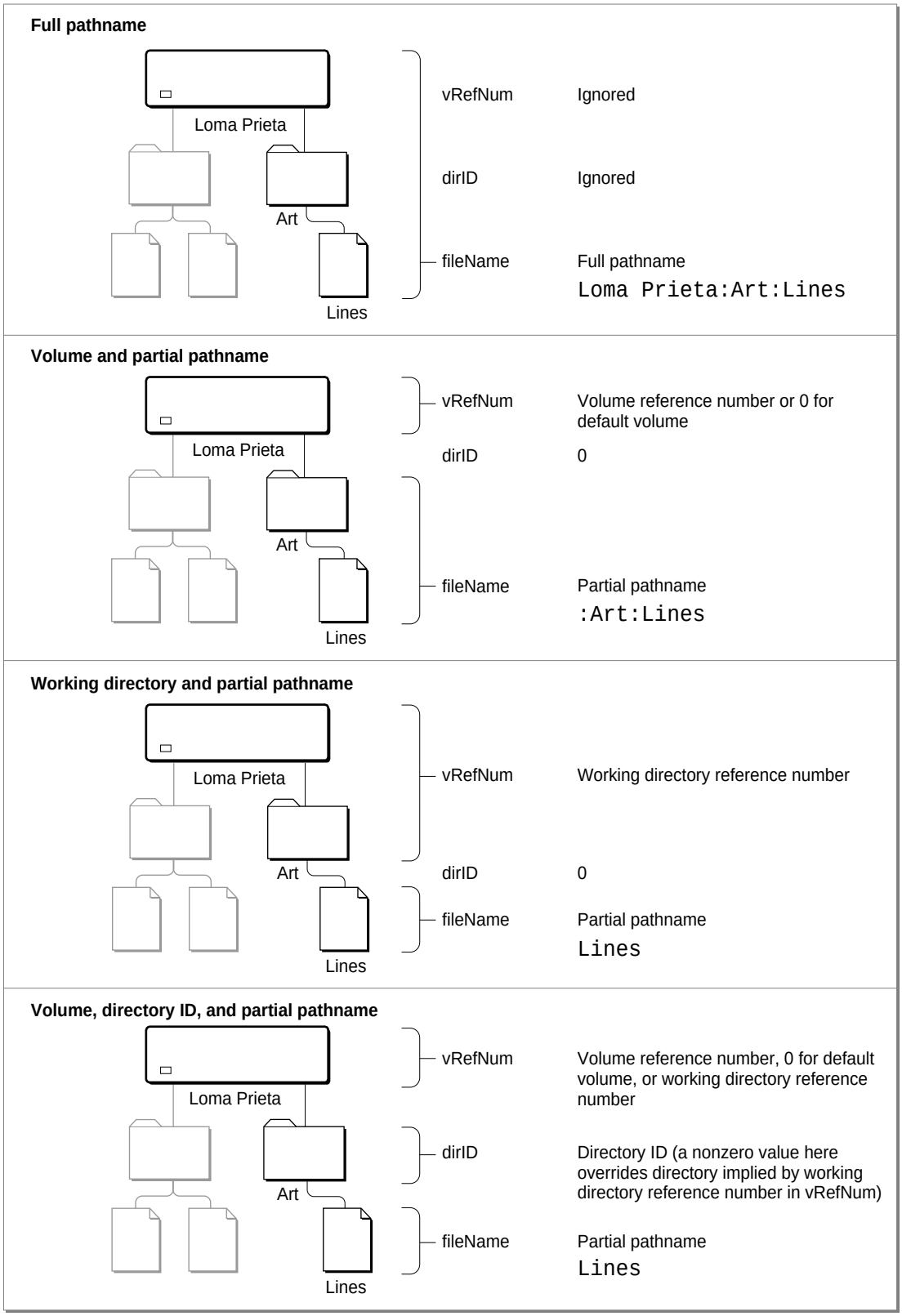
A full pathname consists of the name of the volume, the names of all directories between the root directory and the target, and the name of the target. A full pathname starts with a character other than a colon and contains at least one colon. If the first character is a colon, or if the pathname contains 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.

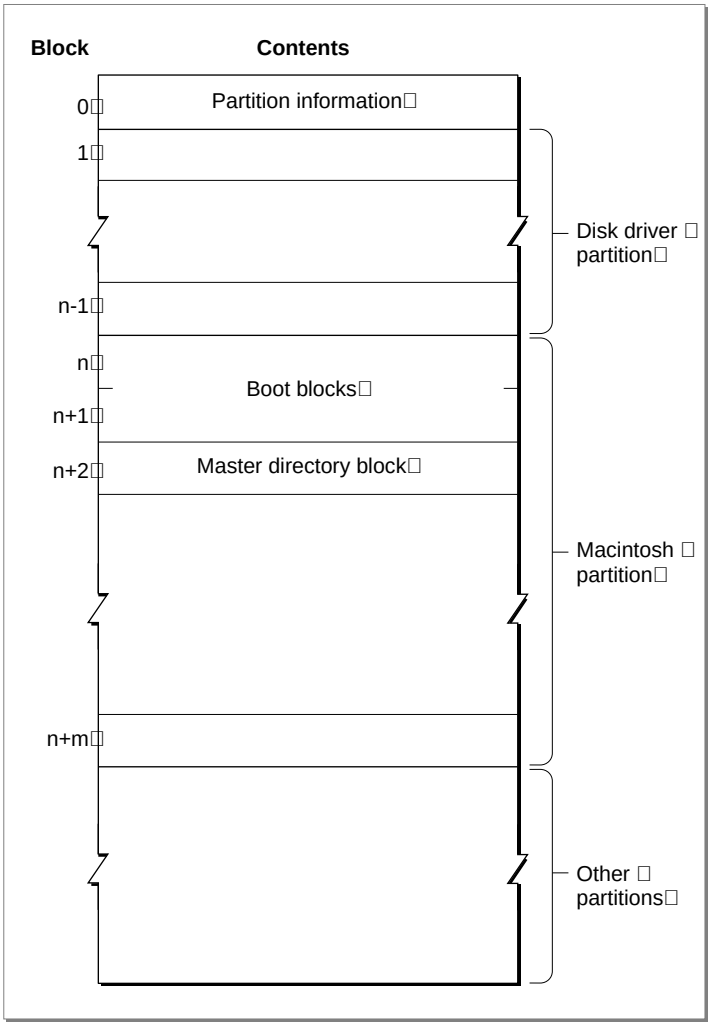
Note
The disk organization illustrated in Figure 2-4 does not apply to Macintosh 3.5-inch floppy disks. Because each floppy disk is one volume, there is no need for a disk partition map. Also, there is no device driver partition on a floppy disk. ◆

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

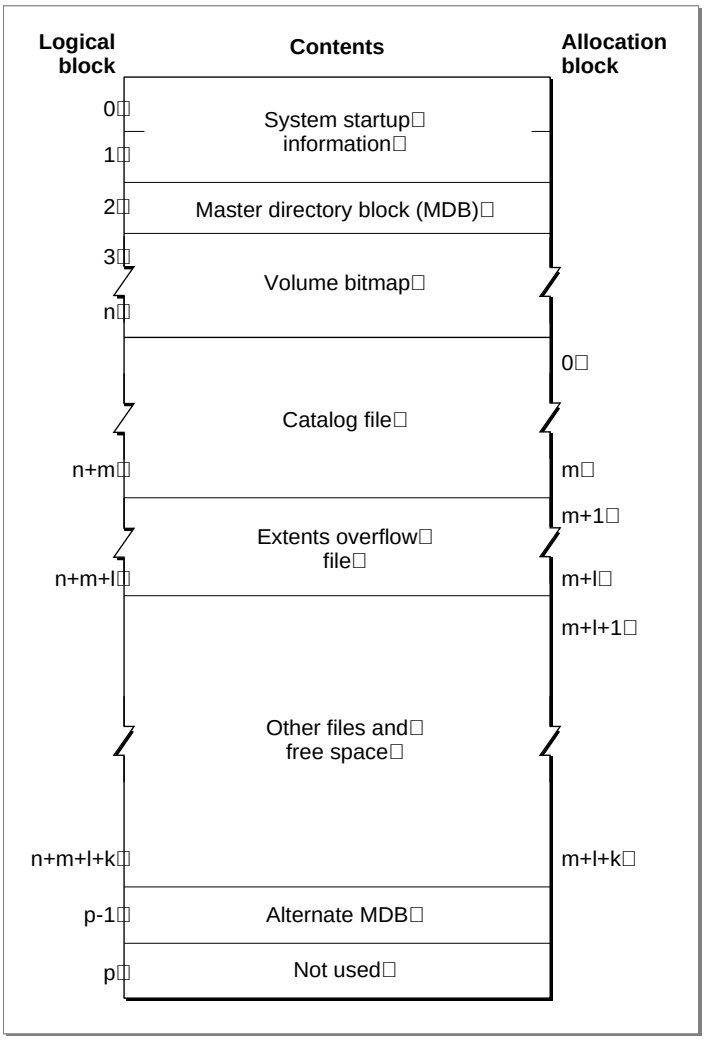


The largest portion of a volume consists of four types of information or areas:

- 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. ◆

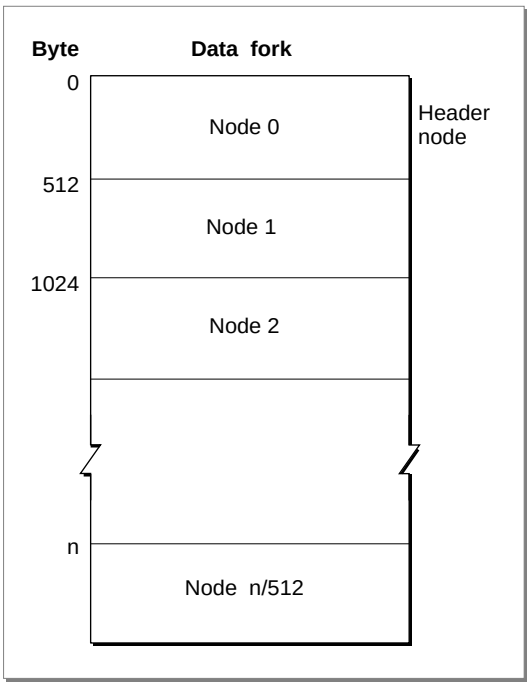
The size of the volume bitmap depends on the number of allocation blocks in the volume, which in turn depends both on the number of physical blocks in the volume and on the size of the volume's allocation blocks (the number of physical blocks per allocation block). For example, a floppy disk that can 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

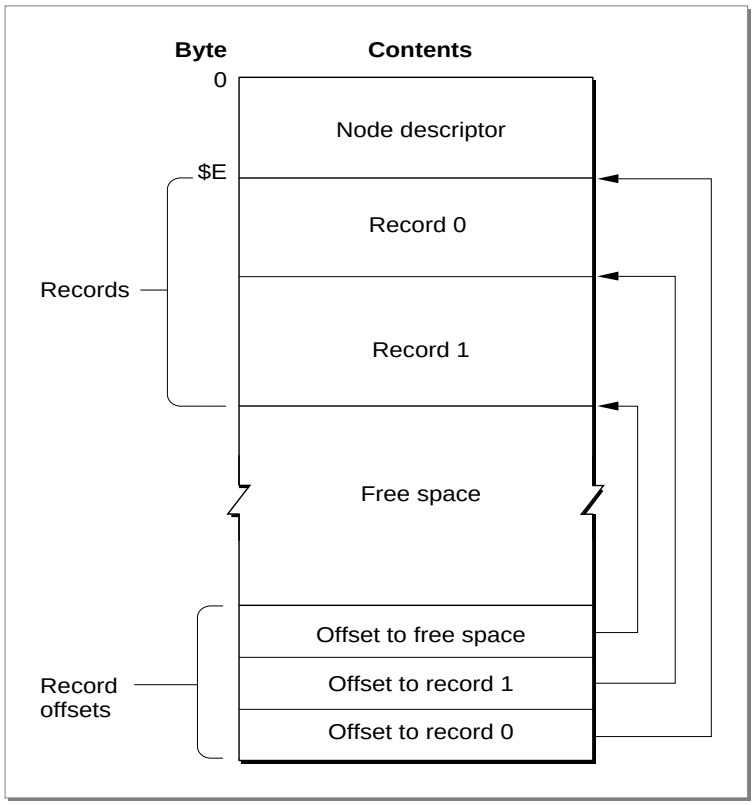
A B*-tree file consists entirely of objects called **nodes,** each of which is 512 bytes long. Figure 2-7 illustrates the structure of a node.

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:

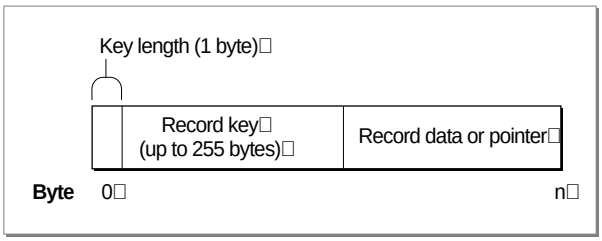
```
CONST                           {node types}
   ndIndxNode     =  $00;      {index node}
   ndHdrNode      =  $01;      {header node}
   ndMapNode      =  $02;      {map node}
   ndLeafNode     =  $FF;      {leaf node}
```

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



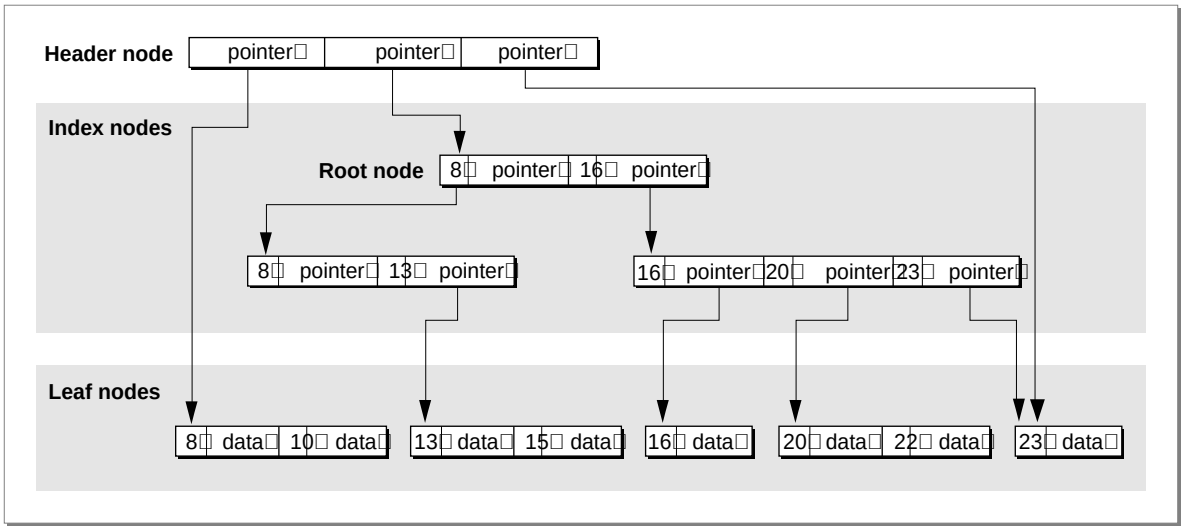
Note
The three records in a B*-tree header node do not have the structure depicted in Figure 2-8. They consist solely of data, as described in the next section, "Header Nodes." Similarly, the single record in a map node consists solely of data; see "Map Nodes" on page 2-70 for details. ◆

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

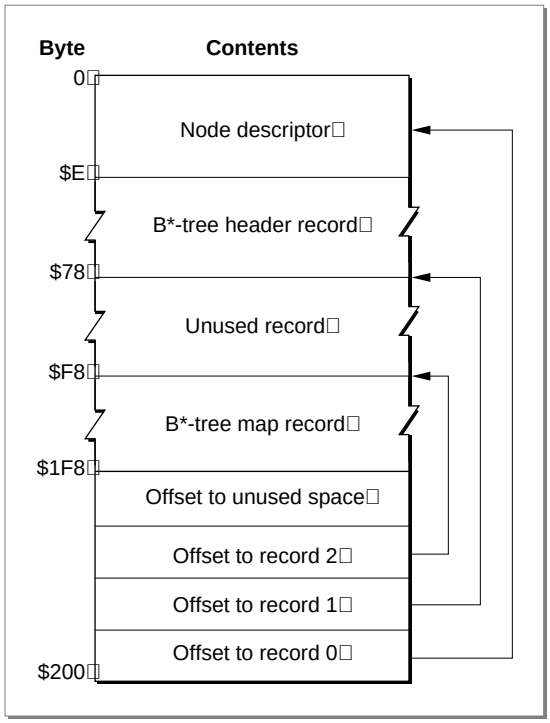
The first node (that is, node 0) in every B*-tree file is a **header node,** which contains essential information about the entire B*-tree file. The File Manager stores the location of the header node of the catalog file in the first 2 bytes of the `drCTExtRec` field of the MDB; the value in those 2 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;
```

Internal Data Types

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