Introduction to File Management

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

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

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

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

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

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

About Files

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

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

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

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

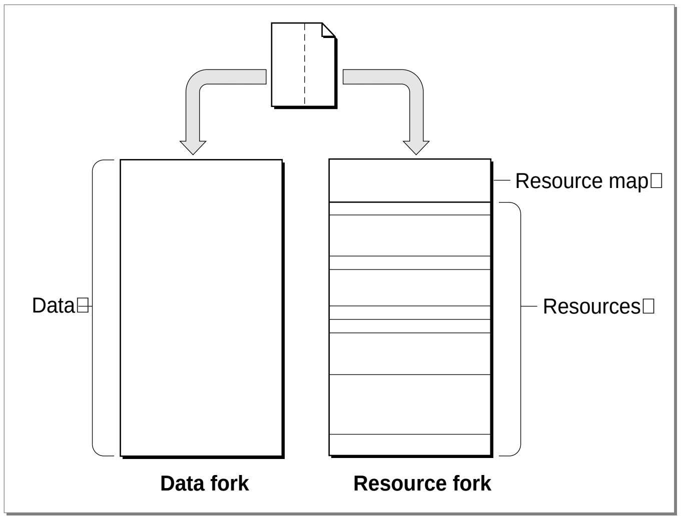
- file forks
- file size and access characteristics
- file system organization
- file naming and identification

File Forks

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

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

Figure 1-1 The two forks of a Macintosh file



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

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

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

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

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

IMPORTANT

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

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

File Size

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

Note

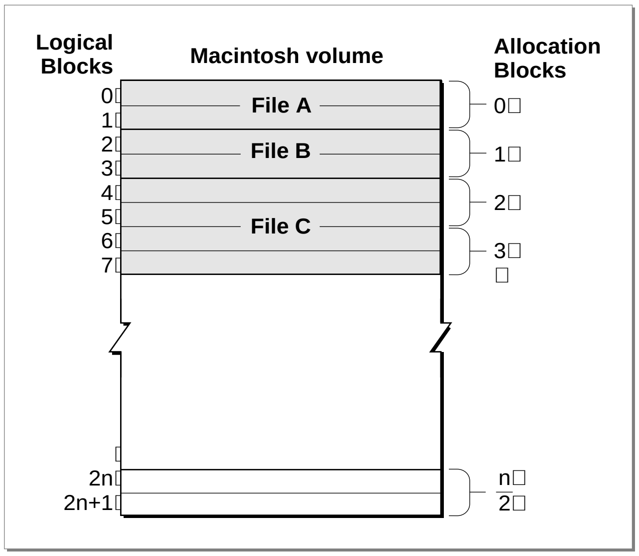
Actually, a file on an HFS volume can be as large as 2 GB ($7FFFFFFF bytes). Most volumes are not large enough to hold a file of that size. An HFS volume currently can be as large as 2 GB. ◆

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

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

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

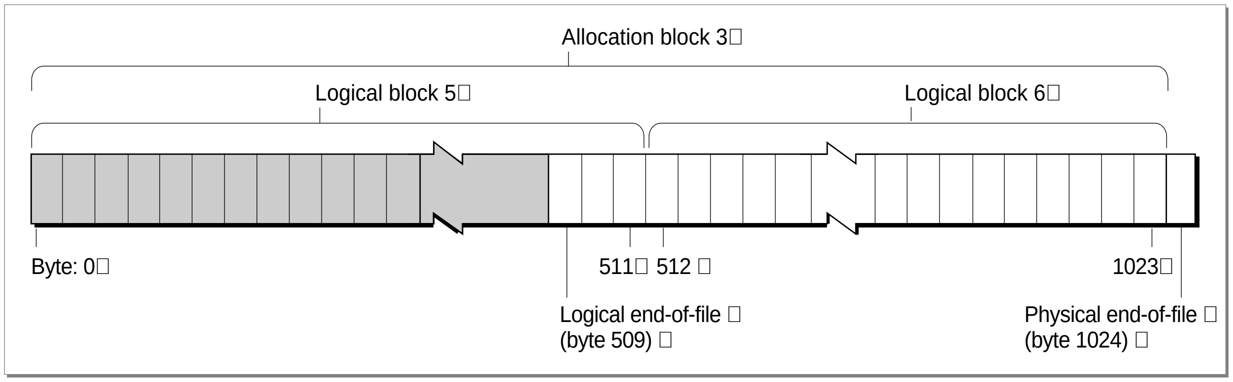
Figure 1-2 Logical blocks and allocation blocks



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

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

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



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

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

File Access Characteristics

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

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

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

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

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

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

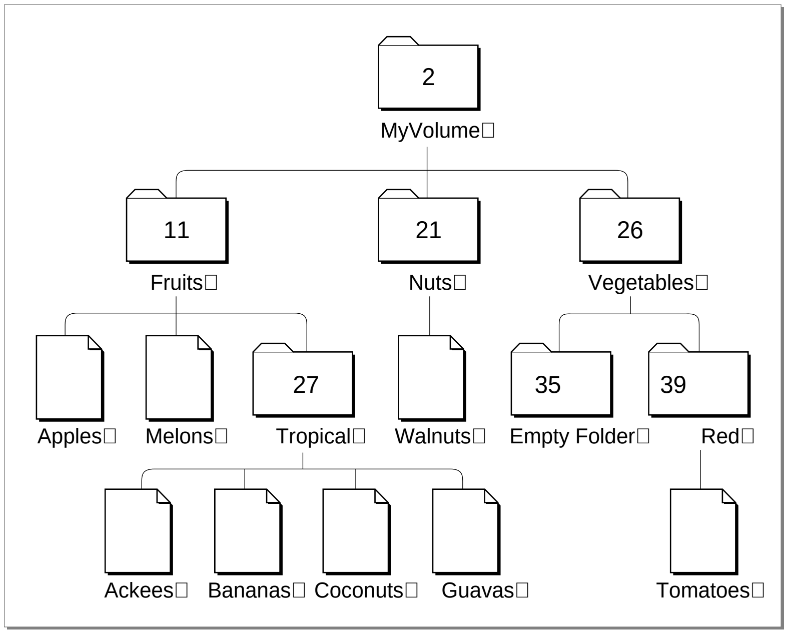
Note

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

The Hierarchical File System

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

Figure 1-4 The Macintosh hierarchical file system



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

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

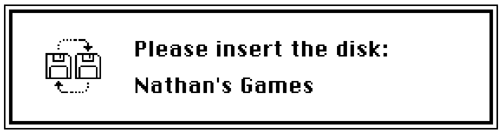
When a volume is mounted, the File Manager assigns a **volume reference number** by which you can refer to the volume for as long as it remains mounted. You can also identify a volume by its **volume name,** a sequence of 1 to 27 printing characters, excluding colons (:). (The File Manager ignores case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

Note

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

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

Figure 1-5 The disk switch dialog box


Please insert the disk:
Nathan's Games

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

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

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

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

Identifying Files and Directories

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

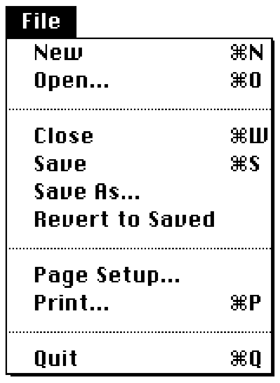
```
TYPE FSSpec    =           {file system specification}
RECORD
   vRefNum:    Integer;    {volume reference number}
   parID:      LongInt;    {directory ID of parent directory}
   name:       Str63;      {filename or directory name}
END;
```

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

Using Files

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

Figure 1-6 A typical File menu



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

Note

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

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

Listing 1-1 Handling the File menu commands

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

Your application should deactivate any menu commands that do not apply to the frontmost window. For example, if the frontmost window is not a document window belonging to your application, then the Close, Save, Save As, and Revert commands should be dimmed when the menu appears. Similarly, if the document in the frontmost window does belong to your application but contains data that has not changed since it was last saved, then the Save menu command should be dimmed. See “Adjusting the File Menu” on page 1-37 for details on implementing this feature. The definitions of the application-defined functions used in Listing 1-1 assume that this feature has been implemented.

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

Testing for File Management Routines

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

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

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

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

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

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

you can write something like

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

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

Note

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

Defining a Document Record

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

Listing 1-3 A sample document record

```
TYPE
   MyDocRecHnd    = ^MyDocRecPtr;
   MyDocRecPtr    = ^MyDocRec;
   MyDocRec       =
   RECORD
      editRec:       TEHandle;         {handle to TextEdit record}
      vScrollBar:    ControlHandle;    {vertical scroll bar}
      hScrollBar:    ControlHandle;    {horizontal scroll bar}
      fileRefNum:    Integer;          {ref num for window's file}
      fileFSSpec:    FSSpec;           {file's FSSpec}
      windowDirty:   Boolean;          {has window data changed?}
   END;
```

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

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

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

Creating a New File

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

Listing 1-4 Handling the New menu command

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

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

Listing 1-5 Creating a new document window

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

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

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

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

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

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

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

Opening a File

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

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

As you can see, it makes sense to divide the process of opening a document into several different routines. You can have a routine that elicits a file selection from the user and another routine that creates a window and reads the file data into it. In the sample listings given here, the function `DoOpenCmd` handles the interaction with the user and `DoOpenFile` reads a file into a new window.

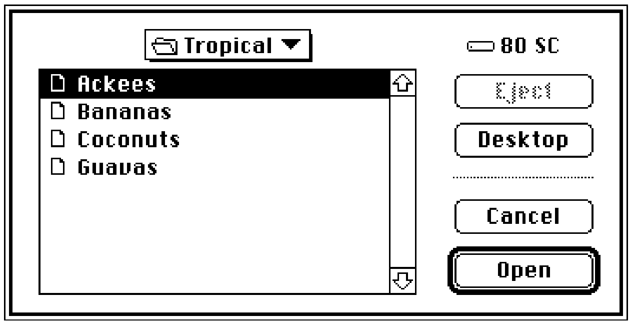
Listing 1-6 shows one way to handle the Open command in the File menu. It uses the Standard File Package routine `StandardGetFile` to determine which file the user wants to open.

Listing 1-6 Handling the Open menu command

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

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

Figure 1-7 The default Open dialog box



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

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

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

Listing 1-7 Opening a file

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

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

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

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

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

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

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

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

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

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

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

Reading File Data

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

Listing 1-8 Reading data from a file

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

   myErr := GetEOF(myFile, myLength);           {get file length}
   myBuffer := NewPtr(myLength);                {allocate a buffer}
   IF myBuffer = NIL THEN
      BEGIN
         DoReadFile := MemError;
         Exit(DoReadFile);
      END;
   myErr := FSRead(myFile, myLength, myBuffer); {read data into buffer}
   IF (myErr = noErr) OR (myErr = eofErr) THEN
      BEGIN                                    {move data into TERec}
         HLock(Handle(myData^^.editRec));
         TESetText(myBuffer, myLength, myData^^.editRec);
         myErr := noErr;
         HUnlock(Handle(myData^^.editRec));
      END;
   DoReadFile := myErr;
END;
```

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

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

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

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

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

Writing File Data

Generally your application writes data to a file in response to the File menu commands Save or Save As. However, your application might also incorporate a scheme that automatically saves all open documents to disk every few minutes. It therefore makes sense to isolate the routines that handle the menu commands from the routines that handle the actual writing of data to disk. This section shows how to write the data stored in a TextEdit record to a file. See "Saving a File" on page 1-26 for instructions on handling the Save and Save As menu commands.

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

Listing 1-9 Writing data into a file

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

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

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

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

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

Listing 1-10 Updating a file safely

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

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

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

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

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

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

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

Saving a File

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

Listing 1-11 Handling the Save menu command

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

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

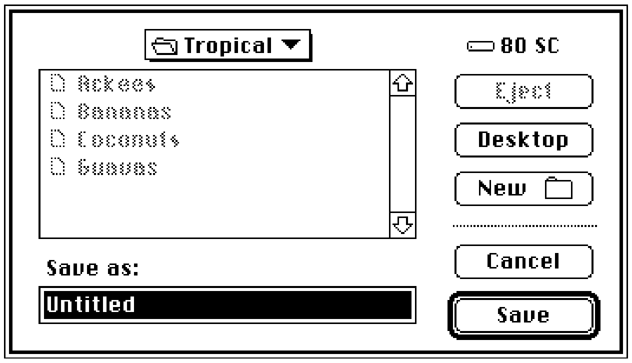
The `DoSaveCmd` function simply checks whether the frontmost window is already associated with a file. If so, then `DoSaveCmd` calls `DoWriteFile` to write the data to disk (using the "safe-save" process illustrated in the previous section). Otherwise, if no file exists for that window, `DoSaveCmd` calls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

Listing 1-12 Handling the Save As menu command

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

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

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

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

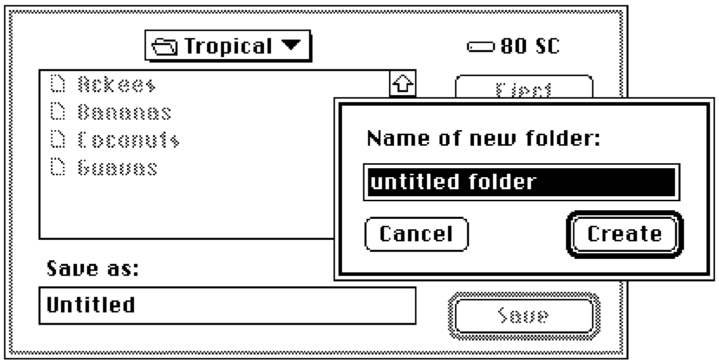
The `StandardPutFile` procedure is similar to the `StandardGetFile` procedure discussed earlier in this chapter. It manages the user interface for the default Save dialog box, illustrated in Figure 1-8.

Figure 1-8 The default Save dialog box



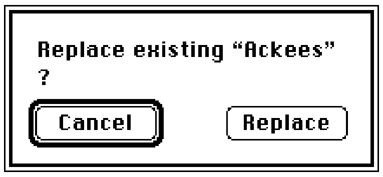
If the user clicks the New Folder button, the Standard File Package presents a subsidiary dialog box like the one shown in Figure 1-9.

Figure 1-9 The new folder dialog box



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

Figure 1-10 The name conflict dialog box



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

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

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

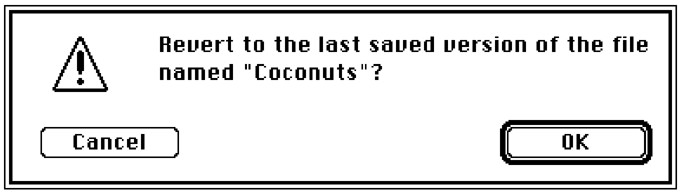
```
FUNCTION DoCopyResource (theType: ResType; theID: Integer;
                        source: Integer; dest: Integer): OSErr;
VAR
   myHandle:   Handle;                        {handle to resource to copy}
   myName:     Str255;                        {name of resource to copy}
   myType:     ResType;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

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

Reverting to a Saved File

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

Figure 1-11 A Revert to Saved dialog box


Revert to the last saved version of the file named "Coconuts"?

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

Listing 1-14 Handling the Revert to Saved menu command

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

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

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

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

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

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

Closing a File

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

Listing 1-15 Handling the Close menu command

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

The `DoCloseCmd` function determines the type of the frontmost window by calling the application-defined function `MyGetWindowType`. (See the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a definition of `MyGetWindowType`.) If the window to be closed is a window belonging to a desk accessory, `DoCloseCmd` closes the desk accessory. If the window to be closed is a dialog window, this procedure just hides the window. If the window to be closed is a document window, `DoCloseCmd` retrieves its document record handle and calls both `DoCloseFile` (defined in Listing 1-16) and `DisposeWindow`. Before you close the file associated with a window, you should check whether the contents of the window have changed since the last time the document was saved. If so, you should ask the user whether to save those changes. Listing 1-16 illustrates one way to do this.

Listing 1-16 Closing a file

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

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

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

Opening Files at Application Startup Time

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

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

IMPORTANT

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

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

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

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

Listing 1-17 Opening files at application launch time

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

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

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

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

Note

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

Using a Preferences File

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

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

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

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

Listing 1-18 Opening a preferences file

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

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

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

   {Read your preference settings here.}

   CloseResFile(myRefNum);
END;
```

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

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

Adjusting the File Menu

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

Listing 1-19 Adjusting the File menu

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

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

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

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

File Management Reference

This section describes the data structures and routines used in this chapter to illustrate basic file management operations. The section “Data Structures” shows the Pascal data structures for the file system specification record and the standard file reply record. The sections that follow describe the Standard File Package routines for opening and saving documents and the File Manager routines for accessing files, manipulating files and directories, accessing volumes, and getting information about documents to be opened when your application is launched.

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

Data Structures

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

File System Specification Record

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

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

Field descriptions

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

Standard File Reply Records

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

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

Field descriptions

`sfGood`	Reports whether the reply record is valid. The value is `TRUE` after the user clicks Save or Open; `FALSE` after the user clicks Cancel. When the user has completed the dialog box, the other fields in the reply record are valid only if the `sfGood` field contains `TRUE`.
`sfReplacing`	Reports whether a file to be saved replaces an existing file of the same name. This field is valid only after a call to the `StandardPutFile` or `CustomPutFile` procedure. When the user assigns a name that duplicates that of an existing file, the Standard File Package asks for verification by displaying a subsidiary dialog box (illustrated in Figure 1-10). If the user verifies the name, the Standard File Package sets the `sfReplacing` field to `TRUE` and returns to your application; if the user cancels the overwriting of the file, the Standard File Package returns to the main dialog box. If the name does not conflict with an existing name, the Standard File Package sets the field to `FALSE` and returns.
`sfType`	Contains the file type of the selected file. (File types are described in the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials*.) Only `StandardGetFile` and `CustomGetFile` return a file type in this field.
`sfFile`	Describes the selected file, folder, or volume with a file system specification record, which contains a volume reference number, parent directory ID, and name. (See the chapter "File Manager" in this book for a complete description of the file system specification record.) If the selected item is an alias for another item, the Standard File Package resolves the alias and places the file system specification record for the target in the `sfFile` field when the user completes the dialog box. If the selected file is a stationery pad, the reply record describes the file itself, not a copy of the file.
`sfScript`	Identifies the script in which the name of the document is to be displayed. (This information is used by the Finder and by the Standard File Package.) A script code of `smSystemScript` (`–1`) represents the default system script.

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

Application Files Records

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

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

Field descriptions

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

File Specification Routines

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

StandardGetFile

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

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

`fileFilter`	A pointer to an optional file filter function, provided by your application, through which `StandardGetFile` passes files of the specified types.
`numTypes`	The number of file types to be displayed. If you specify a `numTypes` value of –1, the first filtering passes files of all types.
`typeList`	A list of file types to be displayed.
`reply`	The reply record, which `StandardGetFile` fills in before returning.

DESCRIPTION

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

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

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

SPECIAL CONSIDERATIONS

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

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

StandardPutFile

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

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

`prompt`	The prompt message to be displayed over the text field.
`defaultName`	The initial name of the file.
`reply`	The reply record, which `StandardPutFile` fills in before returning.

DESCRIPTION

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

SPECIAL CONSIDERATIONS

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

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

File Access Routines

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

Reading, Writing, and Closing Files

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

FSRead

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

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

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

DESCRIPTION

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

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

RESULT CODES

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

FSWrite

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

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

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

DESCRIPTION

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

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

RESULT CODES

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

FSClose

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

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

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

DESCRIPTION

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

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

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

RESULT CODES

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

Manipulating the File Mark

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

GetFPos

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

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

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

DESCRIPTION

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

RESULT CODES

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

SetFPos

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

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

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

DESCRIPTION

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

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

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

RESULT CODES

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

Manipulating the End-of-File

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

GetEOF

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

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

`refNum`	The file reference number of an open file.
`logEOF`	On output, the logical end-of-file.

DESCRIPTION

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

RESULT CODES

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

SEE ALSO

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

SetEOF

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

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

`refNum`	The file reference number of an open file.
`logEOF`	The logical end-of-file.

DESCRIPTION

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

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

RESULT CODES

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

SEE ALSO

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

File and Directory Manipulation Routines

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

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

Opening, Creating, and Deleting Files

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

FSpOpenDF

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

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

`spec`	An `FSSpec` record specifying the file whose data fork is to be opened.
`permission`	A constant indicating the desired file access permissions.
`refNum`	A reference number of an access path to the file's data fork.

DESCRIPTION

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

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

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

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

RESULT CODES

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

FSpCreate

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

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

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

DESCRIPTION

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

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

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

Note

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

RESULT CODES

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

FSpDelete

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

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

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

DESCRIPTION

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

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

RESULT CODES

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

Exchanging the Data in Two Files

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

FSpExchangeFiles

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

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

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

DESCRIPTION

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

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

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

RESULT CODES

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

Creating File System Specifications

The `FSMakeFSSpec` function allows you to create `FSSpec` records.

FSMakeFSSpec

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

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

`vRefNum`	A volume specification. This parameter can contain a volume reference number, a working directory reference number, a drive number, or 0 (to specify the default volume).
`dirID`	A directory specification. This parameter usually specifies the parent directory ID of the target object. If the directory is sufficiently specified by either the `vRefNum` or `fileName` parameter, `dirID` can be set to 0. If you explicitly specify `dirID` (that is, if it has any value other than 0), and if `vRefNum` specifies a working directory reference number, `dirID` overrides the directory ID included in `vRefNum`. If the `fileName` parameter contains an empty string, `FSMakeFSSpec` creates an `FSSpec` record for a directory specified by either the `dirID` or `vRefNum` parameter.
`fileName`	A full or partial pathname. If `fileName` specifies a full pathname, `FSMakeFSSpec` ignores both the `vRefNum` and `dirID` parameters. A partial pathname might identify only the final target, or it might include one or more parent directory names. If `fileName` specifies a partial pathname, then `vRefNum`, `dirID`, or both must be valid.
`spec`	A file system specification to be filled in by `FSMakeFSSpec`.

DESCRIPTION

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

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

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

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

RESULT CODES

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

Volume Access Routines

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

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

Updating Volumes

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

FlushVol

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

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

`volName`	A pointer to the name of a mounted volume.
`vRefNum`	A volume reference number, a working directory reference number, a drive number, or 0 for the default volume.

DESCRIPTION

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

RESULT CODES

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

Obtaining Volume Information

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

GetVInfo

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

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

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

DESCRIPTION

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

RESULT CODES

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

GetVRefNum

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

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

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

DESCRIPTION

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

RESULT CODES

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

Application Launch File Routines

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

Note

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

GetAppParms

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

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

apName	On output, the name of the calling application.
apRefNum	On output, the reference number of the application's resource file.
apParam	On output, a handle to the Finder information about files to open or print.

DESCRIPTION

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

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

SPECIAL CONSIDERATIONS

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

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

ASSEMBLY-LANGUAGE INFORMATION

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

CountAppFiles

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

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

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

DESCRIPTION

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

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

GetAppFiles

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

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

`index`	The index of the file whose information is returned.
`theFile`	A structure containing the returned information.

DESCRIPTION

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

ClrAppFiles

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

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

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

DESCRIPTION

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

Summary of File Management

Pascal Summary

Constants

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

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

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

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

Data Types

File System Specification Record

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

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

Standard File Reply Record

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

Application Files Record

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

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

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

File Specification Routines

Opening Files

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

Saving Files

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

File Access Routines

Reading, Writing, and Closing Files

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

Manipulating the File Mark

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

Manipulating the End-of-File

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

File and Directory Manipulation Routines

Opening, Creating, and Deleting Files

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

Exchanging the Data in Two Files

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

Creating File System Specifications

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

Volume Access Routines

Updating Volumes

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

Obtaining Volume Information

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

Application Launch File Routines

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

C Summary

Constants

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

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

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

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

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

Data Types

File System Specification Record

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

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

Standard File Reply Record

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

typedef struct StandardFileReply StandardFileReply;
```

Application Files Record

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

typedef struct AppFile AppFile;
```

Standard File Type List

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

Callback Routine Pointer Types

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

File Specification Routines

Opening Files

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

Saving Files

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

File Access Routines

Reading, Writing, and Closing Files

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

Manipulating the File Mark

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

Manipulating the End-of-File

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

File and Directory Manipulation Routines

Opening, Creating, and Deleting Files

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

Exchanging the Data in Two Files

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

Creating File System Specifications

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

Volume Access Routines

Updating Volumes

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

Obtaining Volume Information

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

Application Launch File Routines

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

Assembly-Language Summary

Global Variables

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

Result Codes

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