Standard File Package

This chapter describes how your application can use the Standard File Package to manage the user interface for naming and identifying files. The Standard File Package displays the dialog boxes that let the user specify the names and locations of files to be saved or opened, and it reports the user's choices to your application.

The Standard File Package supports both standard and customized dialog boxes. The standard dialog boxes are sufficient for applications that do not require additional controls or other elements in the user interface. The chapter "Introduction to File Management" earlier in this book provides a detailed description of how to display the standard dialog boxes by calling two of the enhanced Standard File Package routines introduced in system software version 7.0. You need to read this chapter if your application needs to use features not described in that earlier chapter (such as customized dialog boxes or a special file filter function). You also need to read this chapter if you want your application to run in an environment where the new routines are not available and your development system does not provide glue code that allows you to call the enhanced routines in earlier system software versions.

To use this chapter, you should be familiar with the Dialog Manager, the Control Manager, and the Finder. You need to know about the Dialog Manager if you want to provide a modal-dialog filter function that handles events received from the Event Manager before they are passed to the `ModalDialog` procedure (which the Standard File Package uses to manage both standard and customized dialog boxes). You need to know about the Control Manager if you want to customize the user interface by adding controls (such as radio buttons or pop-up menus). You need to know about the Finder if your application supports stationery documents. See the appropriate chapters in *Inside Macintosh: Macintosh Toolbox Essentials* for specific information about these system software components.

This chapter provides an introduction to the Standard File Package and then discusses

- how you can display the standard file selection dialog boxes
- how the Standard File Package interprets user actions in those dialog boxes
- how to manage customized dialog boxes
- how to set the directory whose contents are listed in a dialog box
- how to allow the user to select a volume or directory
- how to use the original Standard File Package routines

About the Standard File Package

Macintosh applications typically have a File menu from which the user can save and open documents, via the Save, Save As, and Open commands. When the user chooses Open to open an existing document, your application needs to determine which document to open. Similarly, when the user chooses Save As, or Save when the document is untitled, your application needs to ask the user for the name and location of the file in which the document is to be saved.

The Standard File Package provides a number of routines that handle the user interface between the user and your application when the user saves or opens a document. It displays dialog boxes through which the user specifies the name and location of the document to be saved or opened. It also allows your application to customize the dialog boxes and, through callback routines, to handle user actions during the dialogs. The Standard File Package procedures return information about the user's choices to your application through a reply record.

The Standard File Package is available in all versions of system software. However, significant improvements were made to the package in system software version 7.0. The Standard File Package in version 7.0 introduces

- a pair of simplified procedures (`StandardGetFile` and `StandardPutFile`) that you call to display and handle the standard Open and Save dialog boxes
- a pair of customizable procedures (`CustomGetFile` and `CustomPutFile`) that you call when you need more control over the interaction
- a new reply record (`StandardFileReply`) that identifies files and folders with a file system specification record and that accommodates the new Finder features introduced in system software version 7.0
- a new layout for the standard dialog boxes

This section describes in detail the standard and customized user interfaces provided by the enhanced Standard File Package in system software version 7.0 and later. If your application is to run in earlier system software versions as well, you should read the section "Using the Original Procedures" on page 3-40.

IMPORTANT

If you use the enhanced routines introduced in system software version 7.0, you must also support the Open Documents Apple event. ▲

Standard User Interfaces

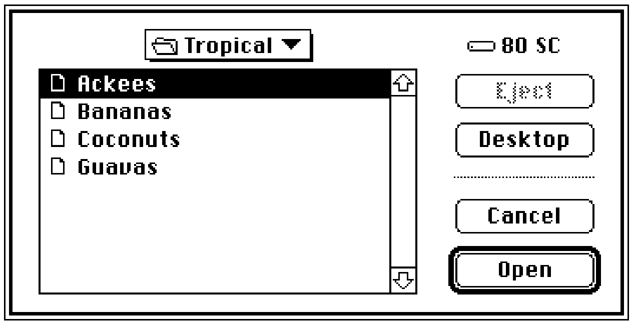
If your application has no special interface requirements, you can use the `StandardGetFile` and `StandardPutFile` procedures to display the standard dialog boxes for opening and saving documents.

Opening Files

You use the `StandardGetFile` procedure when you want to let the user select a file to be opened. Figure 3-1 illustrates a sample dialog box displayed by `StandardGetFile`.

The directory whose contents are listed in the **display list** in the dialog box displayed by `StandardGetFile` is known as the **current directory.** In Figure 3-1, the current directory is named "Tropical." The user can change the current directory in several ways. To ascend the directory hierarchy from the current directory, the user can click the directory pop-up menu and select a new directory from among those in the menu. To ascend one level of the directory hierarchy, the user can click the volume icon. To ascend immediately to the top of the directory hierarchy, the user can click the Desktop button.

Figure 3-1 The default Open dialog box



To descend the directory hierarchy, the user can double-click any of the folder names in the list (or select a folder by clicking its name once and then clicking the Open button). Whenever the current directory changes, the list of folders and files is updated to reflect the contents of the new current directory.

The volume on which the current directory is located is the **current volume** (or **current disk**), whose name is displayed to the right of the directory pop-up menu. If the current volume is a removable volume, the Eject button is active. The user can click Eject to eject the current volume and insert another, which then becomes the current volume. If the user inserts an uninitialized or otherwise unreadable disk, the Standard File Package calls the Disk Initialization Manager to provide the standard user interface for initializing and naming a disk. See the chapter "Disk Initialization Manager" in this book for details.

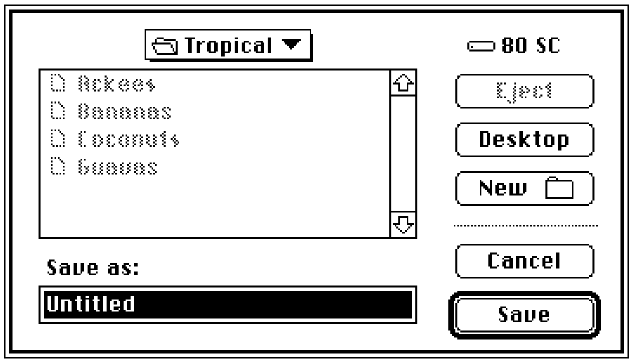
Note that the list of files and folders always contains all folders in the current directory, but it might not contain all files in the current directory. When you call `StandardGetFile`, you can supply a list of the file types that your application can open. The `StandardGetFile` procedure then displays only files of the specified types. You can also supply your own file filter function to help determine which files are displayed. (See "Writing a File Filter Function" on page 3-20 for details.)

When the user is opening a document, `StandardGetFile` interprets some keystrokes as selectors in the displayed list. If the user presses A, for example, `StandardGetFile` selects the first item in the list that starts with the letter *a* (or, if no items in the list start with the letter *a*, the item that starts with the letter closest to *a*). The Standard File Package sets a timer on keystrokes: keystrokes in rapid succession form a string; keystrokes spaced in time are processed separately. See "Keyboard Equivalents" on page 3-7 for a complete list of keyboard equivalents recognized by `StandardGetFile`.

Saving Files

You use the `StandardPutFile` procedure when you want to let the user specify a name and location for a file to be saved. Figure 3-2 illustrates a sample dialog box displayed by `StandardPutFile`.

Figure 3-2 The default Save dialog box

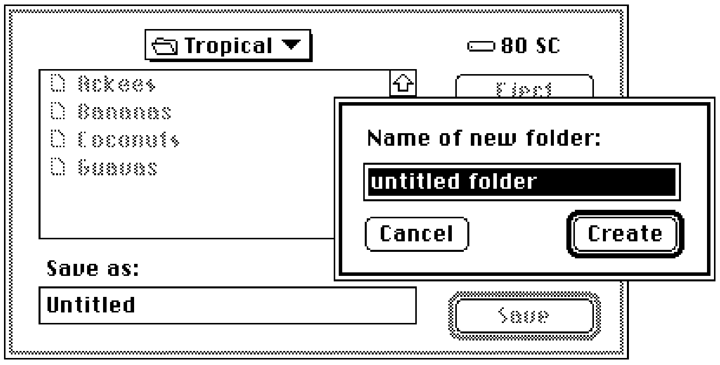


The dialog box displayed by `StandardPutFile` is similar to that displayed by `StandardGetFile`, but includes three additional items. The Save dialog box includes a filename field in which the user can type the name under which to save the file. This filename field is a TextEdit field that permits all the standard editing operations (cut, copy, paste, and so forth). Above the filename field is a line of text specified by your application.

When the user is saving a document, `StandardPutFile` can direct keystrokes to either of two targets: the filename field or the displayed list. When the dialog box first appears, keystrokes are directed to the filename field. If the user presses the Tab key or clicks to select an item in the displayed list, subsequent keystrokes are interpreted as selectors in the displayed list. Each time the user presses the Tab key, keyboard input shifts between the two targets.

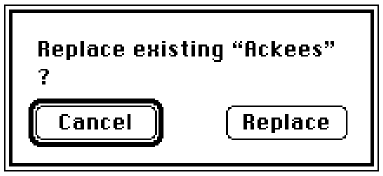
The third additional item in the Save dialog is the New Folder button. When the user clicks the New Folder button, the Standard File Package presents a subsidiary dialog box like the one shown in Figure 3-3.

Figure 3-3 The New Folder dialog box



If the user asks to save a file under a name that already exists at the specified location, the Standard File Package displays a subsidiary dialog box to verify that the new file should replace the existing file, as illustrated in Figure 3-4.

Figure 3-4 The name conflict dialog box


Replace existing "Ackees" ?

The `StandardGetFile` and `StandardPutFile` procedures always display the new dialog boxes. The procedures available before version 7.0 (`SFGetFile`, `SFPutFile`, `SFPGetFile`, and `SFPPutFile`) also display the new dialog boxes when running in version 7.0, unless your application has customized the dialog box. For more details on how the version 7.0 Standard File Package handles earlier procedures, see "Using the Original Procedures" on page 3-40.

Keyboard Equivalents

The Standard File Package recognizes a long list of keyboard equivalents during dialogs.

Keystrokes	**Action**
Up Arrow	Scroll up (backward) through displayed list
Down Arrow	Scroll down (forward) through displayed list
Command–Up Arrow	Display contents of parent directory
Command–Down Arrow	Display contents of selected directory or volume
Command–Left Arrow	Display contents of previous volume
Command–Right Arrow	Display contents of next volume
Command–Shift–Up Arrow	Display contents of desktop
Command-Shift-1	Eject disk in drive 1
Command-Shift-2	Eject disk in drive 2
Tab	Move to next keyboard target
Return *or* Enter	Invoke the default option for the dialog box (Open or Save)
Escape *or* Command-.	Cancel
Command-O	Open the selected item
Command-D	Display contents of desktop
Command-N	Create a new folder
Option-Command-O *or* Option-[click Open]	Select the target of the selected alias item instead of opening it

When the user uses a keyboard equivalent to select a button in the dialog box, the button blinks.

Customized User Interfaces

The standard user interfaces provided by the `StandardGetFile` and `StandardPutFile` procedures might not be adequate for the needs of certain applications. To handle such cases, the Standard File Package provides several routines that you can use to present a customized user interface when opening or saving files. This section gives some simple examples of how you might want to customize the user interfaces and suggests some guidelines you should follow when doing so.

IMPORTANT

You should alter the standard user interfaces only if necessary. Apple Computer, Inc., does not guarantee future compatibility for your application if you use a customized dialog box. ▲

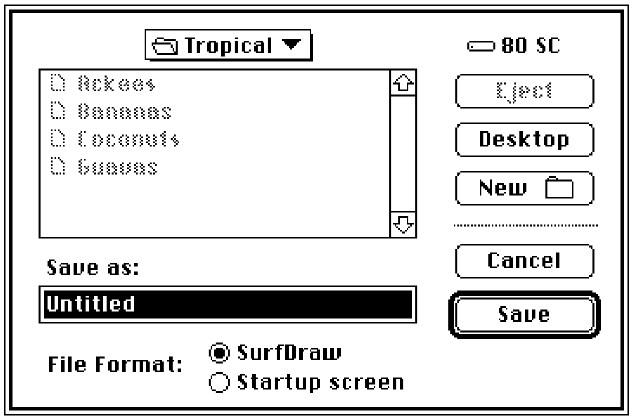
Saving Files

Perhaps the most common reason to customize one of the Standard File Package dialog boxes is to allow the user to save a document in one of several file formats supported by the application. For example, a word-processing application might let the user save a document in the application's own format, in an interchange format, as a file of type `'TEXT'`, and so on.

It is usually best to allow the user to select a file format from within the dialog box displayed in response to a Save or Save As menu command. To do this, you need to add items to the standard dialog box and process user actions in those new items.

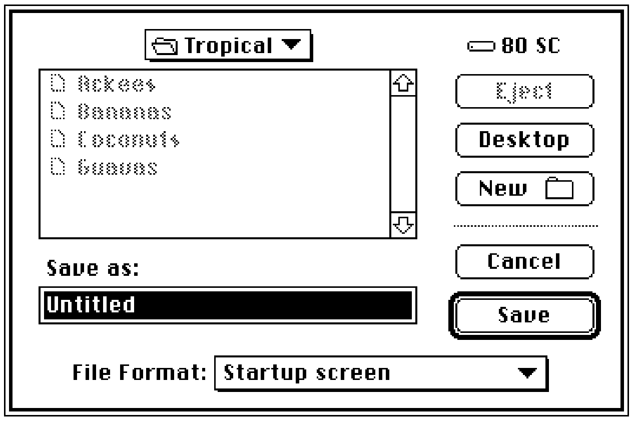
If your application supports only a few file formats, you could simply add the required number of radio buttons to the standard dialog box, as illustrated in Figure 3-5. The application presenting this dialog box supports only two file formats, its own proprietary format (SurfDraw) and the format used for startup screens.

Figure 3-5 The Save dialog box customized with radio buttons



If your application supports more than a couple of alternate file formats, you could add a pop-up menu, as shown in Figure 3-6.

Figure 3-6 The Save dialog box customized with a pop-up menu



Save as: Untitled

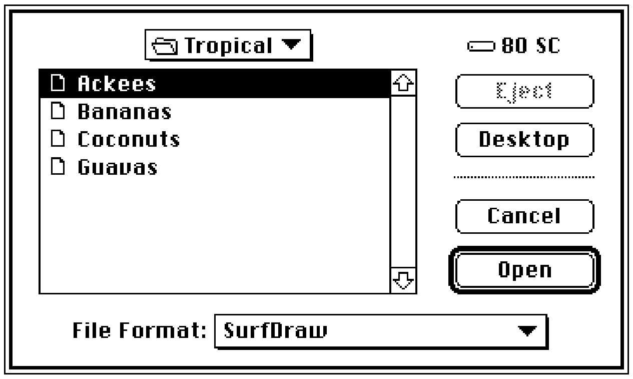
File Format: Startup screen

Opening Files

Your application might also allow the user to open a number of different types of files. In this case, there is less need to customize the Open dialog box than the Save dialog box because you can simply list all the kinds of files your application supports. To avoid clutter in the list of files and folders, however, you might wish to filter out all but one of those types. In this way, the user can dynamically select which type of file to view in the list.

Once again, you might accomplish this by adding radio buttons or a pop-up menu to the Open dialog box, depending on the number of different file types your application supports. Figure 3-7 illustrates a customized Open dialog box that contains a pop-up menu. Only files of the indicated type (and, of course, folders) appear in the list of items available to open.

Figure 3-7 The Open dialog box customized with a pop-up menu



File Format: SurfDraw

For details on some techniques you can use to add items to the standard user interface and process user actions with those additional items, see "Customizing the User Interface" on page 3-16. Note in particular that Listing 3-3, Listing 3-8, and Listing 3-9 together provide a fairly complete implementation of the pop-up menu illustrated in Figure 3-7.

Note
Remember that the user might also open one of your application's documents from the Finder (by double-clicking its icon, for example). As a result, you should in general avoid customizing the Open dialog box for files. ◆

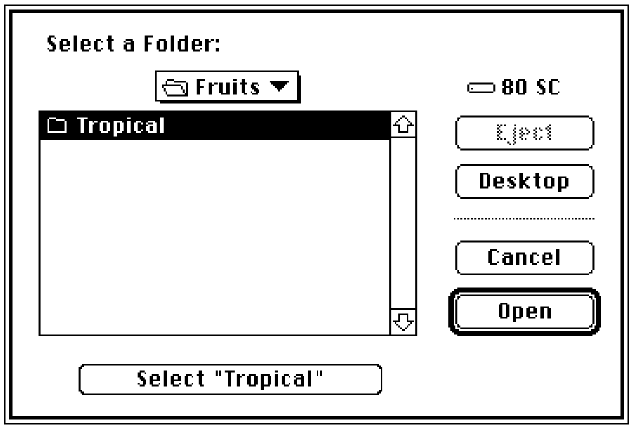
Selecting Volumes and Directories

Sometimes you need to allow the user to select a directory or a volume, not a file. For example, the user might want to select a directory as a first step in searching all the files in the directory for some important information. Similarly, the user might need to select a volume before backing up all the files on that volume.

The standard Open dialog box, however, is designed for selecting files, not volumes or directories. When the user selects a volume or directory from the items in the displayed list and clicks the Open button, the volume or directory is opened and its contents are displayed in the list. The standard Open dialog boxes provide no obvious mechanism for choosing a selected directory instead of opening it.

To allow the user to select a directory—including the volume's root directory, the volume itself—you can add an additional button to the standard Open dialog box. By clicking this button, the user can select a highlighted directory, not open it. This button gives the user an obvious way to select a directory while preserving the well-known mechanism for opening directories to search for the desired directory. Figure 3-8 illustrates the standard Open dialog box modified to include a Select button and a prompt informing the user of the type of action required.

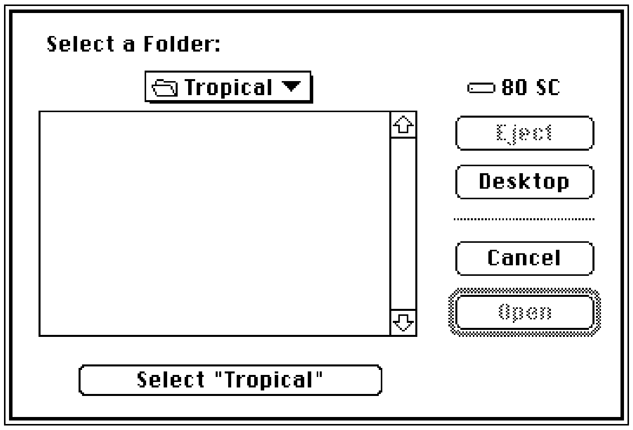
Figure 3-8 The Open dialog box customized to allow selection of a directory



The Select button should display the name of the directory that is selected if the user clicks the button. This, together with the prompt displayed at the top of the dialog box, helps the user differentiate this directory selection dialog box from the standard file opening dialog box. All the other items in the dialog box should maintain their standard appearance and behavior. Any existing keyboard equivalents (in particular, the use of Return and Enter to select the default button) should be preserved. Command-S is recommended as a keyboard equivalent for the new Select button, paralleling the use of Command-D to select the Desktop button and Command-O to select the Open button.

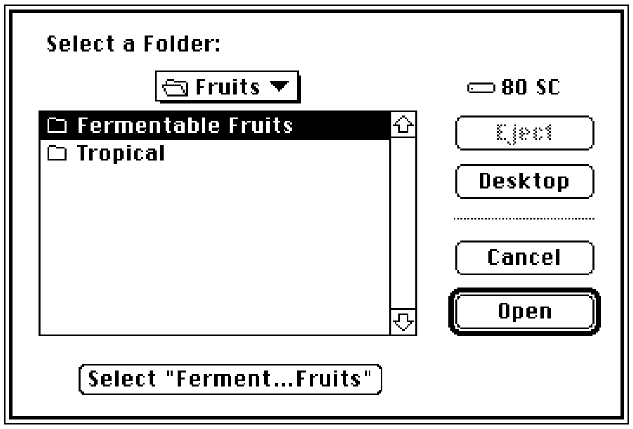
To help maintain consistency among applications using this scheme for selecting directories, your application should open the folder displayed in the pop-up menu if there is no selected item and the user clicks the Select button. In addition, you should disable the Open button if no directory is currently selected. Figure 3-9 illustrates the recommended appearance of the directory selection dialog box in this case.

Figure 3-9 The Open dialog box when no directory is selected



If the name of the directory is too long to fit in the Select button, you should abbreviate the name using an ellipsis character, as shown in Figure 3-10.

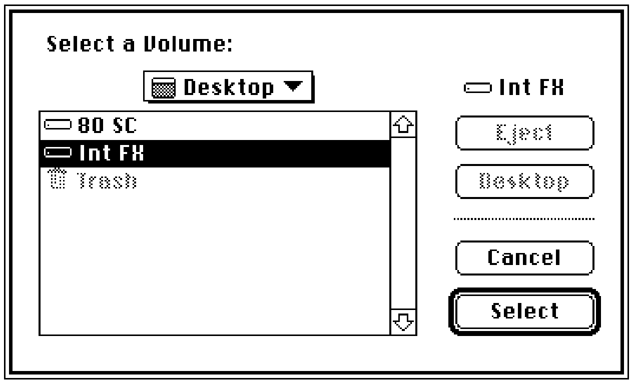
Figure 3-10 The Open dialog box with a long directory name abbreviated



See “Selecting a Directory” beginning on page 3-34 for details on how you can create and manage a directory selection dialog box.

The directory selection dialog boxes illustrated here allow the user to specify the root directory in a volume, which effectively selects the volume itself. However, you might want to limit the user’s selections to the available volumes. To do that, you can create a volume selection dialog box, shown in Figure 3-11.

Figure 3-11 A volume selection dialog box



Notice that the volume selection dialog box uses a prompt specific to selecting a volume and that the Open button is now a Select button. There is no need for a separate Select button, because the user should not be allowed to open any of the listed volumes. (For this same reason, the pop-up menu should not pop up if clicked.) See “Selecting a Volume” on page 3-38 for instructions on implementing a volume selection dialog box.

User Interface Guidelines

In general, you should customize the user interface only if necessary. If you do modify the standard dialog boxes presented by the Standard File Package, you should keep these user interface guidelines in mind:

- Customize a dialog box only by adding items to the standard dialog boxes. Avoid removing existing items from the standard boxes or altering the operation of existing items. In particular, you should avoid modifying the keyboard equivalents recognized by the Standard File Package.
- Add only those items that are necessary for your application to complete the requested action successfully. Avoid adding items that provide unnecessary information or items that provide no information at all (such as logos, icons, or other “window-dressing”).
- Whenever possible, use controls such as radio buttons or pop-up menus whose effects are visible within the dialog box itself. Avoid controls whose use calls subsidiary modal dialog boxes that the user must dismiss before continuing.
- Use controls or other items that are already familiar to the user. Avoid using customized controls that are not also used elsewhere in your application.

Your overriding concern should be to make the customized file identification dialog boxes in your application as similar to the standard dialog boxes as possible while providing the additional capabilities you need.

Using the Standard File Package

You use the Standard File Package to handle the user interface when the user must specify a file to be saved or opened. You typically call the Standard File Package after the user chooses Save, Save As, or Open from the File menu.

When saving a document, you call one of the `PutFile` procedures; when opening a document, you call one of the `GetFile` procedures. The Standard File Package in version 7.0 introduces two pairs of enhanced procedures:

- `StandardPutFile` and `StandardGetFile`, for presenting the standard interface
- `CustomPutFile` and `CustomGetFile`, for presenting a customized interface

Before calling the enhanced Standard File Package procedures, verify that they are available by calling the `Gestalt` function with the `gestaltStandardFileAttr` selector. If `Gestalt` sets the `gestaltStandardFile58` bit in the reply, the four enhanced procedures are available.

If the enhanced procedures are not available, you need to use the original Standard File Package procedures that are available in all system software versions:

- `SFPutFile` and `SFGetFile`, for presenting the standard interface
- `SFPPutFile` and `SFPGetFile`, for presenting a customized interface

This section focuses on the enhanced procedures introduced in system software version 7.0. If you need to use the original procedures, see "Using the Original Procedures" on page 3-40. You can adapt most of the techniques shown in this section for use with the original procedures. In general, however, the original procedures are slightly harder to use and somewhat less powerful than their enhanced counterparts.

All the enhanced procedures return the results of the dialog boxes in a new reply record, `StandardFileReply`.

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

The reply record identifies selected files with a file system specification (`FSSpec`) record. You can pass the `FSSpec` record directly to the File Manager functions that recognize `FSSpec` records, such as `FSpOpenDF` or `FSpCreate`. The reply record also contains additional fields that support the Finder features introduced in system software version 7.0.

The `sfGood` field reports whether the reply record is valid—that is, whether your application can use the information in the other fields. The field is set to `TRUE` after the user clicks Save or Open, and to `FALSE` after the user clicks Cancel.

Your application needs to look primarily at the `sfFile` and `sfReplacing` fields when the `sfGood` field contains `TRUE`. The `sfFile` field contains a file system specification record that describes the selected file or folder. If the selected file is a stationery pad, the reply record describes the file itself, not a copy of the file.

The `sfReplacing` field reports whether a file to be saved replaces an existing file of the same name. This field is valid only after a call to the `StandardPutFile` or `CustomPutFile` procedure. Your application can rely on the value of this field instead of checking for and handling name conflicts itself.

Note
See "Enhanced Standard File Reply Record" on page 3-42 for a complete description of the fields of the `StandardFileReply` record. ◆

The Standard File Package fills in the reply record and returns when the user completes one of its dialog boxes—either by selecting a file and clicking Save or Open, or by clicking Cancel. Your application checks the values in the reply record to see what action to take, if any. If the selected item is an alias for another item, the Standard File Package resolves the alias and places a file system specification record for the target in the `sfFile` field when the user completes the dialog box. (See the chapter "Finder Interface" of *Inside Macintosh: Macintosh Toolbox Essentials* for a description of aliases.)

Presenting the Standard User Interface

You can use the standard dialog boxes provided by the Standard File Package to prompt the user for the name of a file to open or a filename and location to use when saving a document. Use `StandardGetFile` to present the standard interface when opening a file and `StandardPutFile` to present the standard interface when saving a file.

Listing 3-1 illustrates how your application can use `StandardGetFile` to elicit a file specification after the user chooses Open from the File menu.

Listing 3-1 Handling the Open menu command

```
FUNCTION DoOpenCmd: OSErr;
VAR
   myReply:    StandardFileReply;   {Standard File reply record}
   myTypes:    SFTypeList;          {types of files to display}
   myErr:      OSErr;
```

```
BEGIN
   myTypes[0] := 'TEXT';            {display text files only}
   StandardGetFile(NIL, 1, myTypes, myReply);
   IF myReply.sfGood THEN
      myErr := DoOpenFile(myReply.sfFile)
   ELSE
      myErr := UsrCanceledErr;
   DoOpenCmd := myErr;
END;
```

If the user dismisses the dialog box by clicking the Open button, the reply record field `myReply.sfGood` is set to `TRUE`; in that case, the function defined in Listing 3-1 calls the application-defined function `DoOpenFile`, passing it the file system specification record contained in the reply record. For a sample definition of the `DoOpenFile` function, see the chapter "Introduction to File Management" in this book.

The third parameter to `StandardGetFile` is a list of file types that are to appear in the list of files and folders; the second parameter is the number of items in that list of file types. The list of file types is of type `SFTypeList`.

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

If you need to display more than four types of files, you can define a new data type that is large enough to hold all the types you need. For example, you can define the data type `MyTypeList` to hold ten file types:

```
TYPE  MyTypeList  =  ARRAY[0..9] OF OSType;
      MyTListPtr  =  ^MyTypeList;
```

Listing 3-2 shows how to call `StandardGetFile` using an expanded type list.

Listing 3-2 Specifying more than four file types

```
FUNCTION DoOpenCmd: OSErr;
VAR
   myReply:    StandardFileReply;   {Standard File reply record}
   myTypes:    MyTypeList;          {types of files to display}
   myErr:      OSErr;
BEGIN
   myTypes[0] := 'TEXT';            {first file type to display}
   {Put other assignments here.}
   myTypes[9] := 'RTFT';            {tenth file type to display}
   StandardGetFile(NIL, 1, MyTListPtr(myTypes)^, myReply);
   IF myReply.sfGood THEN
      myErr := DoOpenFile(myReply.sfFile)
   ELSE
      myErr := UsrCanceledErr;
   DoOpenCmd := myErr;
END;
```

Note
To display all file types in the dialog box, pass –1 as the second parameter. Invisible files and folders are not shown in the dialog box unless you pass –1 in that parameter. If you pass –1 as the second parameter when calling `CustomGetFile`, the dialog box also lists folders; this is not true when you call `StandardGetFile`. ◆

The first parameter passed to `StandardGetFile` is the address of a file filter function, a function that helps determine which files appear in the list of files to open. (In Listing 3-1, this address is `NIL`, indicating that all files of the specified type are to be listed.) See "Writing a File Filter Function" on page 3-20 for details on defining a filter function for use with `StandardGetFile`.

Customizing the User Interface

If your application requires it, you can customize the user interface for identifying files. To customize a dialog box, you should

- design your dialog box and create the resources that describe it
- write callback routines, if necessary, to process user actions in the dialog box
- call the Standard File Package using the `CustomPutFile` and `CustomGetFile` procedures, passing the resource IDs of the customized dialog boxes and pointers to the callback routines

Depending on the level of customizing you require in your dialog box, you may need to write as many as four different callback routines:

- a file filter function for determining which files the user can open
- a dialog hook function for handling user actions in the dialog boxes
- a modal-dialog filter function for handling user events received from the Event Manager
- an activation procedure for highlighting the display when keyboard input is directed at a customized field defined by your application

To provide the interface illustrated in Figure 3-7, for example, you could replace the definition of `DoOpenCmd` given earlier in Listing 3-1 by the definition given in Listing 3-3.

In addition to the information passed to `StandardGetFile`, `CustomGetFile` requires the resource ID of the customized dialog box, the location of the dialog box on the screen, and pointers to any callback routines and private data you are using.

Listing 3-3 Presenting a customized Open dialog box

```
FUNCTION DoOpenCmd: OSErr;
VAR
   myReply:    StandardFileReply;   {Standard File reply record}
   myTypes:    SFTypeList;          {types of files to display}
   myPoint:    Point;               {upper-left corner of box}
   myErr:      OSErr;
CONST
   kCustomGetDialog = 4000;         {resource ID of custom dialog}
BEGIN
   myErr := noErr;
   SetPt(myPoint, -1, -1);          {center the dialog}
   myTypes[0] := 'SRFD';            {SurfDraw files}
   myTypes[1] := 'STUP';            {startup screens}
   myTypes[2] := 'PICT';            {picture files}
   myTypes[3] := 'RTFT';            {rich text format}

   CustomGetFile(@MyCustomFileFilter, 4, myTypes, myReply,
                 kCustomGetDialog, myPoint, @MyDlgHook,
                 NIL, NIL, NIL, NIL);
   IF myReply.sfGood THEN
      myErr := DoOpenFile(myReply.sfFile);
   DoOpenCmd := myErr;
END;
```

In Listing 3-3, `CustomGetFile` is passed two callback routines, a file filter function (`MyCustomFileFilter`) and a dialog hook function (`MyDlgHook`). See Listing 3-8 (page 3-21) and Listing 3-9 (page 3-27) for sample definitions of these functions.

You can also supply data of your own to the callback routines through a new parameter, `yourDataPtr`, which you pass to `CustomGetFile` and `CustomPutFile`.

Customizing Dialog Boxes

To describe a dialog box, you supply a `'DLOG'` resource that defines the box itself and a `'DITL'` resource that defines the items in the dialog box.

Listing 3-4 shows the resource definition of the default Open dialog box, in Rez input format. (Rez is the resource compiler provided with Apple's Macintosh Programmer's Workshop [MPW]. For a description of Rez format, see the manual that accompanies the MPW software, *MPW: Macintosh Programmer's Development Environment.*)

Listing 3-4 The definition of the default Open dialog box

```
resource 'DLOG' (-6042, purgeable)
   {
      {0, 0, 166, 344}, dBoxProc, invisible, noGoAway, 0,
       –6042, "", noAutoCenter
   };
```

Listing 3-5 shows the resource definition of the default Save dialog box, in Rez input format.

Listing 3-5 The definition of the default Save dialog box

```
resource 'DLOG' (-6043, purgeable)
   {
      {0, 0, 188, 344}, dBoxProc, invisible, noGoAway, 0,
       –6043, "", noAutoCenter
   };
```

Note
You can also use the stand-alone resource editor ResEdit, available from Apple Computer, Inc., or other resource-editing utilities available from third-party developers to create customized dialog box and dialog item list resources. ◆

You must provide an item list (in a `'DITL'` resource with the ID specified in the `'DLOG'` resource) for each dialog box you define. Add new items to the end of the default lists. `CustomGetFile` expects the first 9 items in a customized dialog box to have the same functions as the corresponding items in the `StandardGetFile` dialog box; `CustomPutFile` expects the first 12 items to have the same functions as the corresponding items in the `StandardPutFile` dialog box. If you want to eliminate one of the standard items from the display, leave it in the item list but place its coordinates outside the bounds of the dialog box rectangle.

Listing 3-6 shows the dialog item list for the default Open dialog box, in Rez input format. See "Writing a Dialog Hook Function" beginning on page 3-21 for a list of the dialog box elements these items represent.

Listing 3-6 The item list for the default Open dialog box

```
resource 'DITL'(-6042)
   {  {
      {135, 252, 155, 332}, Button { enabled, "Open" },
      {104, 252, 124, 332}, Button { enabled, "Cancel" },
      {0, 0, 0, 0}, HelpItem { disabled, HMScanhdlg {-6042}},
      {8, 235, 24, 337}, UserItem { enabled },
      {32, 252, 52, 332}, Button { enabled, "Eject" },
      {60, 252, 80, 332}, Button { enabled, "Desktop" },
      {29, 12, 159, 230}, UserItem { enabled },
      {6, 12, 25, 230}, UserItem { enabled },
      {91, 251, 92, 333}, Picture  { disabled, 11 },
   }  };
```

Listing 3-7 shows the dialog item list for the default Save dialog box, in Rez input format.

Listing 3-7 The item list for the default Save dialog box

```
resource 'DITL'(-6043)
   {  {
      {161, 252, 181, 332}, Button { enabled, "Save" },
      {130, 252, 150, 332}, Button { enabled, "Cancel" },
      {0, 0, 0, 0}, HelpItem { disabled, HMScanhdlg {-6043}},
      {8, 235, 24, 337}, UserItem { enabled },
      {32, 252, 52, 332}, Button { enabled, "Eject" },
      {60, 252, 80, 332}, Button { enabled, "Desktop" },
      {29, 12, 127, 230}, UserItem { enabled },
      {6, 12, 25, 230}, UserItem { enabled },
      {119, 250, 120, 334}, Picture { disabled, 11 },
      {157, 15, 173, 227}, EditText { enabled, "" },
      {136, 15, 152, 227}, StaticText { disabled, "Save as:" },
      {88, 252, 108, 332}, UserItem { disabled },
   }  };
```

The third item in each list (`HelpItem`) supplies Apple's Balloon Help for items in the dialog box. This third item specifies the resource ID of the `'hdlg'` resource that contains the help strings for the standard dialog items. If you want to modify the help text of an existing dialog item, you should copy the original `'hdlg'` resource from the System file into your application's resource fork and modify the text in the copied resource as desired; then you must change the resource ID specified in `HelpItem` to the resource ID of the copied and modified resource. To provide Balloon Help for your own items, supply a second `'hdlg'` resource and reference it with another help item at the end of the list. The existing items retain their default text (unless you change that text, as described).

The default dialog item lists used by the original Standard File Package routines do not contain help items, but the Standard File Package does provide Balloon Help when you call those routines in system software version 7.0 and later. If you call one of the original routines and the specified dialog item list does not contain any help items, the Standard File Package uses its default help text for the standard dialog items. If, however, the dialog item list does contain a help item, the Standard File Package assumes that your application provides the text for all help items, including the standard dialog items.

Note
The default Standard File Package dialog boxes support color. The System file contains `'dctb'` resources with the same resource IDs as the default dialog boxes, so that the Dialog Manager uses color graphics ports for the default dialog boxes. (See the chapter “Dialog Manager” of *Inside Macintosh: Macintosh Toolbox Essentials* for a description of the `'dctb'` resource.) If you create your own dialog boxes, include `'dctb'` resources. ◆

Writing a File Filter Function

A **file filter function** determines which files appear in the displayed list when the user is opening a file. Both `StandardGetFile` and `CustomGetFile` recognize file filter functions.

When the Standard File Package is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application’s specifications. (Your application can specify which file types are to be displayed through the `typeList` parameter to either `StandardGetFile` or `CustomGetFile`, as described in “Presenting the Standard User Interface” beginning on page 3-14.) If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

The file filter function receives a pointer to the file’s catalog information record (described in the chapter “File Manager” in this book). The function evaluates the catalog entry and returns a Boolean value that determines whether the file is filtered (that is, a value of `TRUE` suppresses display of the filename, and a value of `FALSE` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

A file filter function to be called by `StandardGetFile` must use this syntax:

```
FUNCTION MyStandardFileFilter (pb: CInfoPBPtr): Boolean;
```

The single parameter passed to your standard file filter function is the address of a catalog information parameter block; see the chapter “File Manager” in this book for a description of the fields of that parameter block.

When `CustomGetFile` calls your file filter function, it can also receive a pointer to any data that you passed in through the call to `CustomGetFile`. A file filter function to be called by `CustomGetFile` must use this syntax:

```
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr):
                               Boolean;
```

Listing 3-8 shows a sample file filter function to be called by `CustomGetFile`. You might define a file filter function like this to support the custom dialog box illustrated in Figure 3-7, which lists files of the type shown in the pop-up box.

Listing 3-8 A sample file filter function

```
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr): Boolean;
BEGIN
   MyCustomFileFilter := TRUE;              {default: don't show the file}
   IF pb^.ioFlFndrInfo.fdType = gTypesArray[gCurrentType] THEN
      MyCustomFileFilter := FALSE;          {show the file}
END;
```

In Listing 3-8, the application global variable `gCurrentType` contains the index in the array `gTypesArray` of the currently selected file type. If the type of a file passed in for evaluation matches the current file type, the filter returns `FALSE`, indicating that `StandardGetFile` should put it in the list. See Listing 3-9 (page 3-27) for an example of how you can use a dialog hook function to change the value of `gCurrentType` according to user selections in the pop-up menu control.

Writing a Dialog Hook Function

A **dialog hook function** handles item selections in a dialog box. It receives a pointer to the dialog record and an item number from the `ModalDialog` procedure via the Standard File Package each time the user selects one of the dialog items. Your dialog hook function checks the item number of each selected item, and then either handles the selection or passes it back to the Standard File Package.

If you provide a dialog hook function, `CustomPutFile` and `CustomGetFile` call your function immediately after calling `ModalDialog`. They pass your function the item number returned by `ModalDialog`, a pointer to the dialog record, and a pointer to the data received from your application, if any. The dialog hook function must use this syntax:

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr;
                    myDataPtr: Ptr): Integer;
```

Your dialog hook function returns as its function result an integer that is either the item number passed to it or some other item number. If it returns one of the item numbers in the following list of constants, the Standard File Package handles the selected item as described later in this section. If your dialog hook function does not handle a selection, it should pass the item number back to the Standard File Package for processing by setting its return value equal to the item number.

```
CONST {items that appear in both the Open and Save dialog boxes}
   sfItemOpenButton        =  1;       {Save or Open button}
   sfItemCancelButton      =  2;       {Cancel button}
   sfItemBalloonHelp       =  3;       {Balloon Help}
   sfItemVolumeUser        =  4;       {volume icon and name}
   sfItemEjectButton       =  5;       {Eject button}
   sfItemDesktopButton     =  6;       {Desktop button}
   sfItemFileListUser      =  7;       {display list}
   sfItemPopUpMenuUser     =  8;       {directory pop-up menu}
   sfItemDividerLinePict   =  9;       {dividing line between buttons}

   {items that appear in Save dialog boxes only}
   sfItemFileNameTextEdit  =  10;      {filename field}
   sfItemPromptStaticText  =  11;      {filename prompt text area}
   sfItemNewFolderUser     =  12;      {New Folder button}
```

You must write your own dialog hook function to handle any items you have added to the dialog box.

Note
The constants that represent disabled items (`sfItemBalloonHelp`, `sfItemDividerLinePict`, and `sfItemPromptStaticText`) have no effect, but they are defined in the header files for the sake of completeness. ◆

The Standard File Package also recognizes a number of constants that do not represent any actual item in the dialog list; these constants are known as **pseudo-items.** There are two kinds of pseudo-items:

- pseudo-items passed to your dialog hook function by the Standard File Package
- pseudo-items passed to the Standard File Package by your dialog hook function

The `sfHookFirstCall` constant is an example of the first kind of pseudo-item. The Standard File Package sends this pseudo-item to your dialog hook function immediately before it displays the dialog box. Your function typically reacts to this item number by performing any necessary initialization.

You can pass back other pseudo-items to indicate that you've handled the user selection or to request some action by the Standard File Package. For example, if the list of files and folders must be rebuilt because of a user selection, you can pass back the pseudo-item `sfHookRebuildList`. Similarly, when your application handles the selection and needs no further action by the Standard File Package, it should return `sfHookNullEvent`. When the dialog hook function passes either `sfHookNullEvent` or an item number that the Standard File Package doesn't recognize, it does nothing.

The Standard File Package recognizes these pseudo-item numbers:

```
CONST {pseudo-items available prior to version 7.0}
   sfHookFirstCall         =  -1;      {initialize display}
   sfHookCharOffset        =  $1000;   {offset for character input}
   sfHookNullEvent         =  100;     {null event}
   sfHookRebuildList       =  101;     {redisplay list}
   sfHookFolderPopUp       =  102;     {display parent-directory menu}
   sfHookOpenFolder        =  103;     {display contents of }
                                       { selected folder or volume}

   {additional pseudo-items introduced in version 7.0}
   sfHookLastCall          =  -2;      {clean up after display}
```

```
sfHookOpenAlias         =  104;     {resolve alias}
sfHookGoToDesktop       =  105;     {display contents of desktop}
sfHookGoToAliasTarget   =  106;     {select target of alias}
sfHookGoToParent        =  107;     {display contents of parent}
sfHookGoToNextDrive     =  108;     {display contents of next drive}
sfHookGoToPrevDrive     =  109;     {display contents of previous drive}
sfHookChangeSelection   =  110;     {select target of reply record}
sfHookSetActiveOffset   =  200;     {switch active item}
```

The Standard File Package uses a set of modal-dialog filter functions (described in "Writing a Modal-Dialog Filter Function" on page 3-28) to map user actions during the dialog onto the defined item numbers. Some of the mapping is indirect. A click of the Open button, for example, is mapped to `sfItemOpenButton` only if a file is selected in the display list. If a folder or volume is selected, the Standard File Package maps the selection onto the pseudo-item `sfHookOpenFolder`.

The lists that follow summarize when various items and pseudo-items are generated and how they are handled. The descriptions indicate the simplest mouse action that generates each item; many of the items can also be generated by keyboard actions, as described in "Keyboard Equivalents" on page 3-7.

Note
Any indicated effects of passing back these constants do not occur until the Standard File Package receives the constant back from your dialog hook function. ◆

Constant descriptions

`sfItemOpenButton`
: Generated when the user clicks Open or Save while a filename is selected. The Standard File Package fills in the reply record (setting `sfGood` to `TRUE`), removes the dialog box, and returns.

`sfItemCancelButton`
: Generated when the user clicks Cancel. The Standard File Package sets `sfGood` to `FALSE`, removes the dialog box, and returns.

`sfItemVolumeUser`
: Generated when the user clicks the volume icon or its name. The Standard File Package rebuilds the display list to show the contents of the folder that is one level up the hierarchy (that is, the parent directory of the current parent directory).

`sfItemEjectButton`
: Generated when the user clicks Eject. The Standard File Package ejects the volume that is currently selected.

`sfItemDesktopButton`
: Generated when the user clicks the Drive button in a customized dialog box defined by one of the earlier procedures. You never receive this item number with the new procedures; when the user clicks the Desktop button, the action is mapped to the item `sfHookGoToDesktop`, described later in this section. The Standard File Package displays the contents of the next drive.

`sfItemFileListUser`
: Generated when the user clicks an item in the display list. The Standard File Package updates the selection and generates this item for your information.

`sfItemPopUpMenuUser`
: Never generated. The Standard File Package's modal-dialog filter function maps clicks on the directory pop-up menu to `sfHookFolderPopUp`, described later in this section.

`sfItemFileNameTextEdit`
: Generated when the user clicks the filename field. TextEdit and the Standard File Package process mouse clicks in the filename field, but the item number is generated for your information.

`sfItemNewFolderUser`
: Generated when the user clicks New Folder. The Standard File Package displays the New Folder dialog box.

The pseudo-items are messages that allow your application and the Standard File Package to communicate and support various features added since the original design of the Standard File Package.

The Standard File Package generates three pseudo-items that give your application the chance to control a customized display.

Constant descriptions

`sfHookFirstCall`
: Generated by the Standard File Package as a signal to your dialog hook function that it is about to display a dialog box. If you want to initialize the display, do so when you receive this item. You can specify the current directory either by returning `sfHookGoToDesktop` or by changing the reply record and returning `sfHookChangeSelection`.

`sfHookLastCall`
: Generated by the Standard File Package as a signal to your dialog hook function that it is about to remove a dialog box. If you created any structures when the dialog box was first displayed, remove them when you receive this item.

`sfHookNullEvent`
: Issued periodically by the Standard File Package if no user action has taken place. Your application can use this null event to perform any updating or periodic processing that might be necessary.

Your application can generate three pseudo-items to request services from the Standard File Package.

Constant descriptions

`sfHookRebuildList`
: Returned by your dialog hook function to the Standard File Package when it needs to redisplay the file list. Your application might need to redisplay the list if, for example, it allows the user to change the file types to be displayed. The Standard File Package rebuilds and displays the list of files that can be opened.

`sfHookChangeSelection`
: Returned by your application to the Standard File Package after your application changes the reply record so that it describes a different file or folder. (You'll need to pass the address of the reply record in the `yourDataPtr` field if you want to do this.) The Standard File Package rebuilds the display list to show the contents of the folder or volume containing the object described in the reply record. It selects the item described in the reply record.

`sfHookSetActiveOffset`
: Your application adds this constant to an item number and sends the result to the Standard File Package. The Standard File Package activates that item in the dialog box, making it the target of keyboard input. This constant allows your application to activate a specific field in the dialog box without explicit input from the user.

The Standard File Package's own modal-dialog filter functions generate a number of pseudo-items that allow its dialog hook functions to support various features introduced since the original design of the standard file dialog boxes. Except under extraordinary circumstances, your dialog hook function always passes any of these item numbers back to the Standard File Package for processing.

Constant descriptions

`sfHookCharOffset`
: The Standard File Package adds this constant to the value of an ASCII character when it's using keyboard input for item selection. The Standard File Package uses the decoded ASCII character to select an entry in the display list.

`sfHookFolderPopUp`
: Generated when the user clicks the directory pop-up menu. The Standard File Package displays the pop-up menu showing all parent directories.

`sfHookOpenFolder`
: Generated when the user clicks the Open button while a folder or volume is selected in the display list. The Standard File Package rebuilds the display list to show the contents of the folder or volume.

`sfHookOpenAlias`
: Generated by the Standard File Package as a signal that the selected item is an alias for another file, folder, or volume. If the selected item is an alias for a file, the Standard File Package resolves the alias, places the file system specification record of the target in the reply record, and returns.

  If the selected item is an alias for a folder or volume, the Standard File Package resolves the alias and rebuilds the display list to show the contents of the alias target.

`sfHookGoToDesktop`
: Generated when the user clicks the Desktop button. The Standard File Package displays the contents of the desktop in the display list.

`sfHookGoToAliasTarget`	Generated when the user presses the Option key while opening an item that is an alias. The Standard File Package rebuilds the display list to display the volume or folder containing the alias target and selects the target.
`sfHookGoToParent`	Generated when the user presses Command–Up Arrow (or clicks the volume icon). The Standard File Package rebuilds the display list to show the contents of the folder that is one level up the hierarchy (that is, the parent directory of the current parent directory).
`sfHookGoToNextDrive`	Generated when the user presses Command–Right Arrow. The Standard File Package displays the contents of the next volume.
`sfHookGoToPrevDrive`	Generated when the user presses Command–Left Arrow. The Standard File Package displays the contents of the previous volume.

The `CustomGetFile` and `CustomPutFile` procedures call your dialog hook function for item selections in both the main dialog box and any subsidiary dialog boxes (such as the dialog box for naming a new folder while saving a document through `CustomPutFile`). To determine whether the dialog record describes the main dialog box or a subsidiary dialog box, check the value of the `refCon` field in the window record in the dialog record.

Note

Prior to system software version 7.0, the Standard File Package did not call your dialog hook function during subsidiary dialog boxes. Dialog hook functions for the new `CustomGetFile` and `CustomPutFile` procedures must check the dialog window's `refCon` field to determine the target of the dialog record. ◆

The defined values for the `refCon` field represent the Standard File dialog boxes.

```
CONST
   sfMainDialogRefCon      =  'stdf';  {main dialog box}
   sfNewFolderDialogRefCon =  'nfdr';  {New Folder dialog box}
   sfReplaceDialogRefCon   =  'rplc';  {name conflict dialog box}
   sfStatWarnDialogRefCon  =  'stat';  {stationery warning}
   sfErrorDialogRefCon     =  'err ';  {general error report}
   sfLockWarnDialogRefCon  =  'lock';  {software lock warning}
```

Constant descriptions

`sfMainDialogRefCon`	The main dialog box, either Open or Save.
`sfNewFolderDialogRefCon`	The New Folder dialog box.
`sfReplaceDialogRefCon`	The dialog box requesting verification for replacing a file of the same name.
`sfStatWarnDialogRefCon`	The dialog box warning that the user is opening the master copy of a stationery pad, not a piece of stationery.

`sfErrorDialogRefCon`	A dialog box reporting a general error.
`sfLockWarnDialogRefCon`	The dialog box warning that the user is opening a locked file and won't be allowed to save any changes.

Listing 3-9 defines a dialog hook function that handles user selections in the customized Open dialog box illustrated in Figure 3-7. Note that this dialog hook function handles selections only in the main dialog box, not in any subsidiary dialog boxes.

Listing 3-9 A sample dialog hook function

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
                   Integer;
VAR
   myType:        Integer;       {menu item selected}
   myHandle:      Handle;        {needed for GetDItem}
   myRect:        Rect;          {needed for GetDItem}
   myIgnore:      Integer;       {needed for GetDItem; ignored}
CONST
   kMyPopUpItem = 10;            {item number of File Type pop-up menu}
BEGIN
   MyDlgHook := item;            {by default, return the item passed in}
   IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
      Exit(MyDlgHook);           {this function is only for main dialog}

   {Do processing of pseudo-items and your own additional item.}
   CASE item OF
      sfHookFirstCall:           {pseudo-item: first time function called}
         BEGIN
            GetDItem(theDialog, kPopUpItem, myType, myHandle, myRect);
            SetCtlValue(ControlHandle(myHandle), gCurrentType);
            MyDlgHook := sfHookNullEvent;
         END;
      kMyPopUpItem:              {user selected File Type pop-up menu}
         BEGIN
            GetDItem(theDialog, item, myIgnore, myHandle, myRect);
            myType := GetCtlValue(ControlHandle(myHandle));
            IF myType <> gCurrentType THEN
               BEGIN
                  gCurrentType := myType;
                  MyDlgHook := sfHookRebuildList;
               END;
         END;
      OTHERWISE
         ;                       {ignore all other items}
   END;
END;
```

The pop-up menu is stored as a control in the application's resource fork. Values stored in the resource determine the appearance of the control, such as the pop-up title text and the menu associated with the control. The Dialog Manager's `ModalDialog` procedure takes care of drawing the box around the pop-up menu and the title of the dialog box. When the dialog hook function is first called, it simply retrieves a handle to that control and sets the value of the pop-up control to the current menu item (stored in the global variable `gCurrentType`). The `MyDlgHook` function then returns `sfHookNullEvent` to indicate that no further processing is required.

When the user clicks the pop-up menu control, `ModalDialog` calls the standard control definition function associated with it. If the user makes a selection in the pop-up menu, `MyDlgHook` is called with the `item` parameter equal to `kPopUpItem`. Your dialog hook function needs simply to determine the current value of the control and respond accordingly. In this case, if the user has selected a new file type, the global variable `gCurrentType` is updated to reflect the new selection, and `MyDlgHook` returns `sfHookRebuildList` to cause the Standard File Package to rebuild the list of files and folders displayed in the dialog box.

For complete details on handling pop-up menus, see the chapters "Control Manager" and "Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Writing a Modal-Dialog Filter Function

A **modal-dialog filter function** controls events closer to their source by filtering the events received from the Event Manager. The Standard File Package itself contains an internal modal-dialog filter function that maps keypresses and other user input onto the equivalent dialog box items. If you also want to process events at this level, you can supply your own filter function.

Note

You can supply a modal-dialog filter function only when you use one of the procedures that displays a customized dialog box (that is, `CustomGetFile`, `CustomPutFile`, `SFPGetFile`, or `SFPPutFile`). ◆

Your modal-dialog filter function determines how the Dialog Manager procedure `ModalDialog` filters events. The `ModalDialog` procedure retrieves events by calling the Event Manager function `GetNextEvent`. As just indicated, the Standard File Package contains an internal filter function that performs some preliminary processing on each event it receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

You might provide a modal-dialog filter function for several reasons. If you have customized the Open or Save dialog boxes by adding one or more items, you might want to map some of the user's keypresses to those items in the same way that the internal filter function maps certain keypresses to existing items.

Another reason to provide a modal-dialog filter function is to avoid a problem that can arise if an update event is received for one of your application's windows while a Standard File Package dialog box is displayed.

Note
The problem described in the following paragraph occurs only in system software versions earlier than version 7.0. The internal modal-dialog filter function installed by the Standard File Package when running in version 7.0 and later avoids the problem by passing the update event to your dialog filter and, if your filter doesn't handle the event, mapping it to a null event. ◆

When `ModalDialog` calls `GetNextEvent` and receives the update event, `ModalDialog` does not know how to respond to it and therefore passes the update event to the Standard File Package's internal filter function. The internal filter function cannot handle the update event either. As a result, if you do not provide your own modal-dialog filter function that handles the update event, that event is never cleared. The next time `ModalDialog` calls `GetNextEvent`, it receives the same update event. `ModalDialog` never receives a null event, so your dialog hook function never performs any processing in response to the `sfHookNullEvent` pseudo-item. You can solve this problem by providing a modal-dialog filter function that handles the update event or changes it to a null event. See Listing 3-10 for details.

A modal-dialog filter function used with `SFPGetFile` and `SFPPutFile` is declared like any filter function passed to `ModalDialog`. Your function is passed a pointer to the dialog record, a pointer to the event record, and the item number. (The modal-dialog filter function is described in the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.)

```
FUNCTION MyModalFilter (theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
```

The modal-dialog filter function used with `CustomGetFile` and `CustomPutFile` requires an additional parameter, a pointer (`myDataPtr`) to the data received from your application, if any.

```
FUNCTION MyModalFilterYD (theDialog: DialogPtr;
                          VAR theEvent: EventRecord;
                          VAR itemHit: Integer;
                          myDataPtr: Ptr): Boolean;
```

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `FALSE`, `ModalDialog` processes the event through its own filters. If your function returns a value of `TRUE`, `ModalDialog` returns with no further action.

The `CustomGetFile` and `CustomPutFile` procedures call your filter function to process events in both the main dialog box and any subsidiary dialog boxes (such as the dialog box for naming a new folder while saving a document through `CustomPutFile`). To determine whether the dialog record describes the main dialog box or a subsidiary dialog box, check the value of the `refCon` field in the window record in the dialog record, as described in "Writing a Dialog Hook Function" beginning on page 3-21.

Listing 3-10 shows how to define a modal-dialog filter function that prevents update events from clogging the event queue.

Listing 3-10 A sample modal-dialog filter function

```
FUNCTION MyModalFilter (theDialog: DialogPtr; VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
BEGIN
   MyModalFilter := FALSE;            {we haven't handled the event yet}
   IF theEvent.what = updateEvt THEN
      IF IsAppWindow(WindowPtr(theEvent.message)) THEN
         BEGIN
            DoUpdateEvent(WindowPtr(theEvent.message));
            MyModalFilter := TRUE;     {we have handled the event}
         END;
END;
```

If this filter function receives an update event for a window other than the Standard File Package dialog box, it calls the application's routine for handling update events (`DoUpdateEvent`) and returns `TRUE` to indicate that the event has been handled. See the chapters "Event Manager" and "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for complete details on handling update events.

Writing an Activation Procedure

The **activation procedure** controls the highlighting of dialog items that are defined by your application and can receive keyboard input. Ordinarily, you need to supply an activation procedure only if your application builds a list from which the user can select entries. The Standard File Package supplies the activation procedure for the file display list and for all TextEdit fields. You can also use the activation procedure to keep track of which field is receiving keyboard input, if your application needs that information.

The target of keyboard input is called the **active field.** The two standard keyboard-input fields are the filename field (present only in Save dialog boxes) and the display list. Unless you override it through your own dialog hook function, the Standard File Package handles the highlighting of its own items and TextEdit fields. When the user changes the keyboard target by pressing the mouse button or the Tab key, the Standard File Package calls your activation procedure twice: the first call specifies which field is being deactivated, and the second specifies which field is being activated. Your application is responsible for removing the highlighting when one of its fields becomes inactive and for adding the highlighting when one of its fields becomes active. The Standard File Package can handle the highlighting of all TextEdit fields, even those defined by your application.

The activation procedure receives four parameters: a dialog pointer, a dialog item number, a Boolean value that specifies whether the field is being activated (`TRUE`) or deactivated (`FALSE`), and a pointer to your own data.

```
PROCEDURE MyActivateProc (theDialog: DialogPtr; itemNo: Integer;
                           activating: Boolean; myDataPtr: Ptr);
```

Setting the Current Directory

The first time your application calls one of the Standard File Package routines, the default current directory (that is, the directory whose contents are listed in the dialog box) is determined by the way in which your application was launched.

- If the user launched your application directly (perhaps by double-clicking its icon in the Finder), the default directory is the directory in which your application is located.
- If the user launched your application indirectly (perhaps by double-clicking one of your application's document icons), the default directory is the directory in which that document is located.

At each subsequent call to one of the Standard File Package routines, the default current directory is simply the directory that was current when the user completed the previous dialog box. You can use the function `GetSFCurDir` defined in Listing 3-11 to determine the current directory.

Listing 3-11 Determining the current directory

```
FUNCTION GetSFCurDir: LongInt;
TYPE
   LongIntPtr = ^LongInt;
CONST
   CurDirStore = $398;
BEGIN
   GetSFCurDir := LongIntPtr(CurDirStore)^;
END;
```

You can use the `GetSFCurVol` function defined in Listing 3-12 to determine the current volume.

Listing 3-12 Determining the current volume

```
FUNCTION GetSFCurVol: Integer;
TYPE
   IntPtr = ^Integer;
CONST
   SFSaveDisk = $214;
BEGIN
   GetSFCurVol := -IntPtr(SFSaveDisk)^;
END;
```

If necessary, you can change the default current directory and volume. For example, when the user needs to select a dictionary file for a spell-checking application, the application might set the current directory to a directory containing document-specific dictionary files. This saves the user from having to navigate the directory hierarchy from the directory containing documents to that containing dictionary files. You can use the procedure `SetSFCurDir` defined in Listing 3-13 to set the current directory.

Listing 3-13 Setting the current directory

```
PROCEDURE SetSFCurDir (dirID: LongInt);
TYPE
   LongIntPtr = ^LongInt;
CONST
   CurDirStore = $398;
BEGIN
   LongIntPtr(CurDirStore)^ := dirID;
END;
```

You can use the procedure `SetSFCurVol` defined in Listing 3-14 to set the current volume.

Listing 3-14 Setting the current volume

```
PROCEDURE SetSFCurVol (vRefNum: Integer);
TYPE
   IntPtr = ^Integer;
CONST
   SFSaveDisk = $214;
BEGIN
   IntPtr(SFSaveDisk)^ := -vRefNum;
END;
```

Note
Most applications don’t need to alter the default current directory or volume. ◆

If you are using the enhanced Standard File Package routines, you can set the current directory by filling in the fields of the file system specification in the reply record passed to `CustomGetFile` or `CustomPutFile`. You do this within your dialog hook function. Listing 3-15 defines a dialog hook function that makes the currently active System Folder the current directory.

Listing 3-15 Setting the current directory

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
                   Integer;
VAR
   myReplyPtr:    StandardFileReplyPtr;
   foundVRefNum:  Integer;
   foundDirID:    LongInt;
   myErr:         OSErr;
BEGIN
   MyDlgHook := item;            {by default, return the item passed in}
   IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
      Exit(MyDlgHook);           {this function is only for main dialog box}

   CASE item OF
      sfHookFirstCall:           {pseudo-item: first time function called}
         BEGIN
            myReplyPtr := StandardFileReplyPtr(myDataPtr);
            myErr := FindFolder(kOnSystemDisk, kSystemFolderType,
                            kDontCreateFolder, foundVRefNum, foundDirID);
            IF myErr = noErr THEN
               BEGIN
                  myReplyPtr^.sfFile.parID := foundDirID;
                  myReplyPtr^.sfFile.vRefNum := foundVRefNum;
                  MyDlgHook := sfHookChangeSelection;
               END;
         END;
      OTHERWISE
         ;                       {ignore all other items}
   END;
END;
```

This dialog hook function installs the System Folder's volume reference number and parent directory ID into the file system specification whose address is passed in the `myDataPtr` parameter. Because the dialog hook function returns the constant `sfHookChangeSelection` the first time it is called (that is, in response to the `sfHookFirstCall` pseudo-item), the Standard File Package sets the current directory to the indicated directory when the dialog box is displayed.

Selecting a Directory

You can present the recommended user interface for selecting a directory by calling the `CustomGetFile` procedure and passing it the addresses of a custom file filter function and a dialog hook function. See "Selecting Volumes and Directories" on page 3-10 for a description of the appearance and behavior of the directory selection dialog box.

The file filter function used to select directories is quite simple; it ensures that only directories, not files, are listed in the dialog box displayed by `CustomGetFile`. Listing 3-16 defines a file filter function you can use for this purpose.

Listing 3-16 A file filter function that lists only directories

```
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr): Boolean;
CONST
   kFolderBit = 4;                 {bit set in ioFlAttrib for a directory}
BEGIN                              {list directories only}
   MyCustomFileFilter := NOT BTst(pb^.ioFlAttrib, kFolderBit);
END;
```

The function `MyCustomFileFilter` simply inspects the appropriate bit in the file attributes (`ioFlAttrib`) field of the catalog information parameter block passed to it. If the directory bit is set, the file filter function returns `FALSE`, indicating that the item should appear in the list; otherwise, the file filter function returns `TRUE` to exclude the item from the list. Because a volume is identified via its root directory, volumes also appear in the list of items in the dialog box.

The title of the Select button should identify which directory is available for selection. You can use the `SetButtonTitle` procedure defined in Listing 3-17 to set the title of a button.

Your dialog hook function calls the `SetButtonTitle` procedure to copy the truncated title of the selected item into the Select button. This title eliminates possible user confusion about which directory is available for selection. If no item in the list is selected, the dialog hook function uses the name of the directory shown in the pop-up menu as the title of the Select button.

Listing 3-17 Setting a button's title

```
PROCEDURE SetButtonTitle (ButtonHdl: Handle; name: Str255; ButtonRect: Rect);
VAR
   result:  Integer;    {result of TruncString}
   width:   Integer;    {width available for name of directory}
BEGIN
   gPrevSelectedName := name;
   WITH ButtonRect DO
      width := (right - left) - (StringWidth('Select ""') + CharWidth(' '));
   result := TruncString(width, name, smTruncMiddle);
   SetCTitle(ControlHandle(ButtonHdl), CONCAT('Select "', name, '"'));
   ValidRect(ButtonRect);
END;
```

The `SetButtonTitle` procedure is passed a handle to the button whose title is to be changed, the name of the directory available for selection, and the button's enclosing rectangle. The global variable `gPrevSelectedName` holds the full directory name, before truncation.

A dialog hook function manages most of the process of letting the user select a director. Listing 3-18 defines a dialog hook function that handles user selections in the dialog box.

Listing 3-18 Handling user selections in the directory selection dialog box

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
                   Integer;
CONST
   kGetDirBTN     = 10;          {Select directory button}
TYPE
   SFRPtr         = ^StandardFileReply;
VAR
   myType:        Integer;       {menu item selected}
   myHandle:      Handle;        {needed for GetDItem}
   myRect:        Rect;          {needed for GetDItem}
   myName:        Str255;
   myPB:          CInfoPBRec;
   mySFRPtr:      SFRPtr;
   myErr:         OSErr;
BEGIN
   MyDlgHook := item;      {default, except in special cases below}
   IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
      Exit(MyDlgHook);     {this function is only for main dialog box}

   GetDItem(theDialog, kGetDirBTN, myType, myHandle, myRect);
   IF item = sfHookFirstCall THEN
```

```
      BEGIN
         {Determine current folder name and set title of Select button.}
         WITH myPB DO
            BEGIN
               ioCompletion := NIL;
               ioNamePtr := @myName;
               ioVRefNum := GetSFCurVol;
               ioFDirIndex := - 1;
               ioDirID := GetSFCurDir;
            END;
         myErr := PBGetCatInfo(@myPB, FALSE);
         SetButtonTitle(myHandle, myName, myRect);
      END
   ELSE
      BEGIN
         {Get mySFRPtr from 3rd parameter to hook function.}
         mySFRPtr := SFRPtr(myDataPtr);
         {Track name of folder that can be selected.}
         IF (mySFRPtr^.sfIsFolder) OR (mySFRPtr^.sfIsVolume) THEN
            myName := mySFRPtr^.sfFile.name
         ELSE
            BEGIN
               WITH myPB DO
                  BEGIN
                     ioCompletion := NIL;
                     ioNamePtr := @myName;
                     ioVRefNum := mySFRPtr^.sfFile.vRefNum;
                     ioFDirIndex := -1;
                     ioDrDirID := mySFRPtr^.sfFile.parID;
                  END;
               myErr := PBGetCatInfo(@myPB, FALSE);
            END;
         {Change directory name in button title as needed.}
         IF myName <> gPrevSelectedName THEN
            SetButtonTitle(myHandle, myName, myRect);

         CASE item OF
            kGetDirBTN:                   {force return by faking a cancel}
               MyDlgHook := sfItemCancelButton;
            sfItemCancelButton:
               gDirSelectionFlag := FALSE;{flag no directory was selected}
            OTHERWISE
               ;
         END; {CASE}
      END;
END;
```

The MyDlgHook dialog hook function defined in Listing 3-18 calls the File Manager function PBGetCatInfo to retrieve the name of the directory to be selected. When the dialog hook function is first called (that is, when item is set to sfHookFirstCall), MyDlgHook determines the current volume and directory by calling the functions GetSFCurVol and GetSFCurDir. When MyDlgHook is called each subsequent time, MyDlgHook calls PBGetCatInfo with the volume reference number and directory ID of the previously opened directory.

When the user clicks the Select button, MyDlgHook returns the item sfItemCancelButton. When the user clicks the real Cancel button, MyDlgHook sets the global variable gDirSelectionFlag to FALSE, indicating that the user didn't select a directory. The function DoGetDirectory uses that variable to distinguish between clicks of Cancel and clicks of Select.

The function DoGetDirectory defined in Listing 3-19 uses the file filter function and the dialog hook functions defined above to manage the directory selection dialog box. On exit, DoGetDirectory returns a standard file reply record describing the selected directory.

Listing 3-19 Presenting the directory selection dialog box

```
FUNCTION DoGetDirectory: StandardFileReply;
VAR
   myReply:          StandardFileReply;
   myTypes:          SFTypeList;         {types of files to display}
   myPoint:          Point;              {upper-left corner of box}
   myNumTypes:       Integer;
   myModalFilter:    ModalFilterYDProcPtr;
   myActiveList:     Ptr;
   myActivateProc:   ActivateYDProcPtr;
   myName:           Str255;
CONST
   rGetDirectoryDLOG = 128;              {resource ID of custom dialog box}
BEGIN
   gPrevSelectedName := '';        {initialize name of previous selection}
   gDirSelectionFlag := TRUE;      {initialize directory selection flag}
   myNumTypes := -1;               {pass all types of files to file filter}
   myPoint.h := -1;                {center dialog box on screen}
   myPoint.v := -1;
   myModalFilter := NIL;
   myActiveList := NIL;
   myActivateProc := NIL;

   CustomGetFile(@MyCustomFileFilter, myNumTypes, myTypes, myReply,
               rGetDirectoryDLOG, myPoint, @MyDlgHook, myModalFilter,
               myActiveList, myActivateProc, @myReply);
   {Get the name of the directory.}
   IF gDirSelectionFlag AND myReply.sfIsVolume THEN
      myName := Concat(myReply.sfFile.name, ':')
   ELSE
      myName := myReply.sfFile.name;

   IF gDirSelectionFlag AND myReply.sfIsVolume THEN
      myReply.sfFile.name := myName
   ELSE IF gDirSelectionFlag THEN
      myReply.sfFile.name := gPrevSelectedName;
   gDirSelectionFlag := FALSE;
   DoGetDirectory := myReply;
END;
```

The `DoGetDirectory` function initializes the two global variables `gPrevSelectedName` and `gDirSelectionFlag`. As you have seen, these two variables are used by the custom dialog hook function. Then `DoGetDirectory` calls `CustomGetFile` to display the directory selection dialog box and handle user selections. When the user selects a directory or clicks the Cancel button, the dialog hook function returns `sfItemCancelButton` and `CustomGetFile` exits. At that point, the reply record contains information about the last item selected in the list of available items.

Selecting a Volume

You can present the recommended user interface for selecting a volume by calling the `CustomGetFile` procedure and passing it the addresses of a custom file filter function and a dialog hook function. See "Selecting Volumes and Directories" on page 3-10 for a description of the appearance and behavior of the volume selection dialog box.

The file filter function used to select volumes is quite simple; it ensures that only volumes, not files or directories, are listed in the dialog box displayed by `CustomGetFile`. Listing 3-16 defines a file filter function you can use to do this.

Listing 3-20 A file filter function that lists only volumes

```
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr): Boolean;
CONST
   kFolderBit = 4;                 {bit set in ioFlAttrib for a directory}
BEGIN                              {list volumes only}
   MyCustomFileFilter := TRUE;     {assume you don't list the item}
   IF BTst(pb^.ioFlAttrib, kFolderBit) AND (pb^.ioDrParID = fsRtParID) THEN
      MyCustomFileFilter := FALSE;
END;
```

The function `MyCustomFileFilter` inspects the appropriate bit in the file attributes (`ioFlAttrib`) field of the catalog information parameter block passed to it. If the directory bit is set, `MyCustomFileFilter` checks whether the parent directory ID of the directory is equal to `fsRtParID`, which is always the parent directory ID of a volume's root directory. If it is, the file filter function returns `FALSE`, indicating that the item should appear in the list of volumes; otherwise, the file filter function returns `TRUE` to exclude the item from the list.

A dialog hook function for handling the items in the volume selection dialog box is defined in Listing 3-21.

Listing 3-21 Handling user selections in the volume selection dialog box

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr; myDataPtr: Ptr):
                   Integer;
VAR
   myType:        Integer;       {menu item selected}
   myHandle:      Handle;        {needed for GetDItem}
   myRect:        Rect;          {needed for GetDItem}
   myName:        Str255;        {new title for Open button}
BEGIN
   MyDlgHook := item;      {default, except in special cases below}
   IF GetWRefCon(WindowPtr(theDialog)) <> LongInt(sfMainDialogRefCon) THEN
      Exit(MyDlgHook);     {this function is only for main dialog box}

   CASE item OF
      sfHookFirstCall:
         BEGIN
            {Set button title and go to desktop.}
            myName := 'Select';
            GetDItem(theDialog, sfItemOpenButton, myType, myHandle, myRect);
            SetCTitle(ControlHandle(myHandle), myName);
            MyDlgHook := sfHookGoToDesktop;
         END;
      sfHookGoToDesktop:         {map Cmd-D to a null event}
         MyDlgHook := sfHookNullEvent;
      sfHookChangeSelection:
         MyDlgHook := sfHookGoToDesktop;
      sfHookGoToNextDrive:       {map Cmd-Left Arrow to a null event}
         MyDlgHook := sfHookNullEvent;
      sfHookGoToPrevDrive:       {map Cmd-Right Arrow to a null event}
         MyDlgHook := sfHookNullEvent;
      sfItemOpenButton, sfHookOpenFolder:
         MyDlgHook := sfItemOpenButton;
      OTHERWISE
         ;
   END;
END;
```

You can prompt the user to select a volume by calling the function `DoGetVolume` defined in Listing 3-22.

Listing 3-22 Presenting the volume selection dialog box

```
FUNCTION DoGetVolume: StandardFileReply;
VAR
   myReply:          StandardFileReply;
   myTypes:          SFTypeList;          {types of files to display}
   myPoint:          Point;               {upper-left corner of box}
   myNumTypes:       Integer;
   myModalFilter:    ModalFilterYDProcPtr;
   myActiveList:     Ptr;
   myActivateProc:   ActivateYDProcPtr;
CONST
   rGetVolumeDLOG    = 129;               {resource ID of custom dialog box}
BEGIN
   myNumTypes := -1;                      {pass all types of files}
   myPoint.h := -1;                       {center dialog box on screen}
   myPoint.v := -1;
   myModalFilter := NIL;
   myActiveList := NIL;
   myActivateProc := NIL;

   CustomGetFile(@MyCustomFileFilter, myNumTypes, myTypes, myReply,
                 rGetVolumeDLOG, myPoint, @MyDlgHook, myModalFilter,
                 myActiveList, myActivateProc, @myReply);

   DoGetVolume := myReply;
END;
```

Using the Original Procedures

The Standard File Package still recognizes all procedures available before system software version 7.0 (`SFGetFile`, `SFPutFile`, `SFPGetFile`, and `SFPPutFile`). It displays the new interface for all applications that don't customize the dialog boxes in incompatible ways (that is, applications that specify both the dialog hook and the modal-dialog filter pointers as `NIL` and that specify no alternative dialog ID).

When the Standard File Package can't use the enhanced dialog box layout because an application customized the dialog box with the earlier procedures, it nevertheless makes some changes to the display:

- It changes the label of the Drive button to Desktop and makes the desktop the root of the display.
- It moves the volume icon slightly to the right, to make room for selection highlighting around the display list field.

If, however, a customized dialog box has suppressed the file display list (by specifying coordinates outside of the dialog box), the Standard File Package uses the earlier interface, on the assumption that the dialog box is designed for volume selection.

If you need to use the procedures available before system software version 7.0, you need to be aware of a number of differences between those procedures and the enhanced procedures. These are the most important differences:

- The original procedures do not recognize some pseudo-items under previous system software versions. For example, the pseudo-item `sfHookLastCall` is not used before version 7.0. See the comments under "Constants" in "Summary of the Standard File Package" (beginning on page 3-60) for information on which pseudo-items are universally available.
- The original standard file reply record (type `SFReply`) returns a working directory reference number, not a volume reference number. Typically, you should immediately convert that number to a volume reference number and directory ID using `GetWDInfo` or `PBGetWDInfo`. Then close the working directory by calling `CloseWD` or `PBCloseWD`. For details on these functions, see the chapter "File Manager" in this book.
- Dialog hook functions used with the original procedures are not passed a `myDataPtr` parameter.

Standard File Package Reference

This section describes the data structures and routines that are specific to the Standard File Package. The "Data Structures" section shows the Pascal data structures for the original and the enhanced Standard File reply records. The section "Standard File Package Routines" describes routines for opening and saving files. The section "Application-Defined Routines" describes the routines that your application can define to customize the operations of the Standard File Package routines.

Data Structures

The Standard File Package exchanges information with your application using a standard file reply record. If you use the procedures introduced in system software version 7.0, you use a reply record of type `StandardFileReply`. If you use the procedures available before version 7.0, you must use a reply record of type `SFReply`.

Enhanced Standard File Reply Record

When you use one of the procedures `StandardPutFile`, `StandardGetFile`, `CustomPutFile`, or `CustomGetFile`, you pass a reply record of type `StandardFileReply`.

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

Field descriptions

`sfGood`	Reports whether the reply record is valid. The value is `TRUE` after the user clicks Save or Open; `FALSE` after the user clicks Cancel. When the user has completed the dialog box, the other fields in the reply record are valid only if the `sfGood` field contains `TRUE`.
`sfReplacing`	Reports whether a file to be saved replaces an existing file of the same name. This field is valid only after a call to the `StandardPutFile` or `CustomPutFile` procedure. When the user assigns a name that duplicates that of an existing file, the Standard File Package asks for verification by displaying a subsidiary dialog box (illustrated in Figure 3-4, page 3-7). If the user verifies the name, the Standard File Package sets the `sfReplacing` field to `TRUE` and returns to your application; if the user cancels the overwriting of the file, the Standard File Package returns to the main dialog box. If the name does not conflict with an existing name, the Standard File Package sets the field to `FALSE` and returns.
`sfType`	Contains the file type of the selected file. (File types are described in the chapter “Finder Interface” in *Inside Macintosh: Macintosh Toolbox Essentials*.) Only `StandardGetFile` and `CustomGetFile` return a file type in this field.
`sfFile`	Describes the selected file, folder, or volume with a file system specification record, which contains a volume reference number, parent directory ID, and name. (See the chapter “File Manager” in this book for a complete description of the file system specification record.) If the selected item is an alias for another item, the Standard

	File Package resolves the alias and places the file system specification record for the target in the `sfFile` field when the user completes the dialog box. If the selected file is a stationery pad, the reply record describes the file itself, not a copy of the file.
`sfScript`	Identifies the script in which the name of the document is to be displayed. (This information is used by the Finder and by the Standard File Package.) A script code of `smSystemScript` (–1) represents the default system script.
`sfFlags`	Contains the Finder flags from the Finder information record in the catalog entry for the selected file. (See the chapter "Finder Interface" in *Inside Macintosh: Macintosh Toolbox Essentials* for a description of the Finder flags.) This field is returned only by `StandardGetFile` and `CustomGetFile`. If your application supports stationery, it should check the stationery bit in the Finder flags to determine whether to treat the selected file as stationery. Unlike the Finder, the Standard File Package does not automatically create a document from a stationery pad and pass your application the new document. If the user opens a stationery document from within an application that does not support stationery, the Standard File Package displays a dialog box warning the user that the master copy is being opened.
`sfIsFolder`	Reports whether the selected item is a folder (`TRUE`) or a file or volume (`FALSE`). This field is meaningful only during the execution of a dialog hook function.
`sfIsVolume`	Reports whether the selected item is a volume (`TRUE`) or a file or folder (`FALSE`). This field is meaningful only during the execution of a dialog hook function.
`sfReserved1`	Reserved.
`sfReserved2`	Reserved.

Original Standard File Reply Record

When you use one of the original Standard File Package procedures `SFPutFile`, `SFGetFile`, `SFPPutFile`, or `SFPGetFile`, you pass a reply record of type `SFReply`.

```
SFReply =
RECORD
   good:       Boolean;    {TRUE if user did not cancel}
   copy:       Boolean;    {reserved}
   fType:      OSType;     {file type}
   vRefNum:    Integer;    {working directory reference number}
   version:    Integer;    {reserved}
   fName:      Str63;      {filename}
END;
```

Field descriptions

`good`	Reports whether the reply record is valid. The value is `TRUE` after the user clicks Save or Open; `FALSE` after the user clicks Cancel. When the user has completed the dialog box, the other fields in the reply record are valid only if the value of `good` is `TRUE`.
`copy`	Reserved.
`fType`	Contains the file type of the selected file. (File types are described in the chapter "Finder Interface" of *Inside Macintosh: Macintosh Toolbox Essentials*.) Only `SFGetFile` and `SFPGetFile` return a file type in this field.
`vRefNum`	Contains the working directory reference number of the selected file.
`version`	Reserved.
`fName`	Contains the name of the selected file.

Note

In spite of its name, the `vRefNum` field does not contain a volume reference number. Instead, it contains a working directory reference number, which encodes both the volume reference number and the parent directory ID of the selected file. You can obtain the volume reference number and directory ID of the file by calling `GetWDInfo` or `PBGetWDInfo`. See the chapter "File Manager" in this book for details about working directory reference numbers. ◆

Standard File Package Routines

This section describes the routines you can use to prompt the user for a file's name and location after a request to save or open a file. If your application is designed to run in system software versions prior to version 7.0, you must use either `SFGetFile` or `SFPGetFile` when opening a file and either `SFPutFile` or `SFPPutFile` when saving a file.

If your application is designed to take advantage of features introduced in system software version 7.0 or later, you can use the new routines intended to simplify the code required to elicit a filename from the user. The `StandardPutFile` and `StandardGetFile` procedures are simplified versions of the original procedures for handling the user interface during the storing and retrieving of files. The `CustomPutFile` and `CustomGetFile` procedures are customizable versions of the same procedures.

Saving Files

You can use the `StandardPutFile` procedure to present the standard user interface when the user asks to save a file. If you need to add elements to the default dialog boxes or exercise greater control over user actions in the dialog box, use `CustomPutFile`.

If your application is designed to execute in system software versions earlier than version 7.0, you can use the corresponding procedures `SFPutFile` and `SFPPutFile`.

StandardPutFile

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

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

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

DESCRIPTION

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `StandardPutFile` are

Trap macro	Selector
`_Pack3`	$0005

SPECIAL CONSIDERATIONS

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

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

CustomPutFile

Use the `CustomPutFile` procedure when your application requires more control over the Save dialog box than is possible using `StandardPutFile`.

```
PROCEDURE CustomPutFile (prompt: Str255; defaultName: Str255;
                           VAR reply: StandardFileReply;
                           dlgID: Integer; where: Point;
                           dlgHook: DlgHookYDProcPtr;
                           filterProc: ModalFilterYDProcPtr;
                           activeList: Ptr;
                           activateProc: ActivateYDProcPtr;
                           yourDataPtr: UNIV Ptr);
```

`prompt`	The prompt message to be displayed over the text field.
`defaultName`	The initial name of the file.
`reply`	The reply record, which `CustomPutFile` fills in before returning.
`dlgID`	The resource ID of a customized dialog template. To use the standard template, set this parameter to 0.
`where`	The upper-left corner of the dialog box, in global coordinates. If you specify the point (–1,–1), `CustomPutFile` automatically centers the dialog box on the screen.
`dlgHook`	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you have not added any items to the dialog box and want the standard items handled in the standard ways. See “Writing a Dialog Hook Function” on page 3-21 for a description of the dialog hook function.
`filterProc`	A pointer to your modal-dialog filter function, which determines how the `ModalDialog` procedure filters events when called by the `CustomPutFile` procedure. Specify a value of `NIL` if you are not supplying your own function. See “Writing a Modal-Dialog Filter Function” on page 3-28 for a description of the modal-dialog filter function.
`activeList`	A pointer to a list of all dialog items that can be activated—that is, can be the target of keyboard input. If you supply an `activeList` parameter of `NIL`, `CustomPutFile` uses the default targets (the filename field and the list of files and folders). If you have added any fields that can accept keyboard input, you must modify the list. The list is stored as an array of 16-bit integers. The first integer is the number of items in the list. The remaining integers are the item numbers of all possible keyboard targets, in the order that they are activated by the Tab key.

`activateProc`
: A pointer to your activation procedure, which controls the highlighting of dialog items that are defined by your application and that can receive keyboard input. See "Writing an Activation Procedure" on page 3-30 for a description of the activation procedure.

`yourDataPtr`
: Any 4-byte value; usually, a pointer to optional data supplied by your application. When `CustomPutFile` calls any of your callback routines, it adds this parameter, making the data available to your callback routines. If you are not supplying any data of your own, you can specify a value of `NIL`.

DESCRIPTION

The `CustomPutFile` procedure is an alternative to `StandardPutFile` when you want to display a customized Save dialog box or handle the default dialog box in a customized way. During the dialog, `CustomPutFile` gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a name and authorizing the save operation or by canceling the save operation. The `CustomPutFile` procedure returns the user's input in a record of type `StandardFileReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `CustomPutFile` are

Trap macro	Selector
`_Pack3`	$0007

SPECIAL CONSIDERATIONS

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

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

SFPutFile

Use the `SFPutFile` procedure to display the standard Save dialog box when the user is saving a file.

```
PROCEDURE SFPutFile (where: Point; prompt: Str255;
                     origName: Str255; dlgHook: DlgHookProcPtr;
                     VAR reply: SFReply);
```

`where`	The upper-left corner of the dialog box, in global coordinates.
`prompt`	The prompt message to be displayed over the text field.
`origName`	The initial name of the file.
`dlgHook`	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you want the standard items handled in the standard ways. See “Writing a Dialog Hook Function” on page 3-21 for a description of the dialog hook function.
`reply`	The reply record, which `SFPutFile` fills in before returning.

DESCRIPTION

The `SFPutFile` procedure presents a dialog box through which the user specifies the name and location of a file to be written to. During the dialog, `SFPutFile` gets and handles events until the user completes the interaction, either by selecting a name and authorizing the save or by canceling the save. The `SFPutFile` procedure returns the user's input in a record of type `SFReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SFPutFile` are

Trap macro	Selector
`_Pack3`	$0001

SPECIAL CONSIDERATIONS

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

SFPPutFile

Use the `SFPPutFile` procedure when your application requires more control over the Save dialog box than is possible using `SFPutFile`.

```
PROCEDURE SFPPutFile (where: Point; prompt: Str255;
                        origName: Str255; dlgHook: DlgHookProcPtr;
                        VAR reply: SFReply; dlgID: Integer;
                        filterProc: ModalFilterProcPtr);
```

`where`	The upper-left corner of the dialog box, in global coordinates.
`prompt`	The prompt message to be displayed over the text field.
`origName`	The initial name of the file, if any.

`dlgHook`	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you have not added any items to the dialog box and want the standard items handled in the standard ways. See "Writing a Dialog Hook Function" on page 3-21 for a description of the dialog hook function.
`reply`	The reply record, which `SFPPutFile` fills in before returning.
`dlgID`	The resource ID of a customized dialog template. To use the standard template, set this parameter to –3999.
`filterProc`	A pointer to your modal-dialog filter function, which determines how the `ModalDialog` procedure filters events when called by the `SFPPutFile` procedure. Specify a value of `NIL` if you are not supplying your own function. See "Writing a Modal-Dialog Filter Function" on page 3-28 for a description of the modal-dialog filter function.

DESCRIPTION

The `SFPPutFile` procedure is an alternative to `SFPutFile` when you want to display a customized Save dialog box or handle the default dialog box in a customized way. During the dialog, `SFPPutFile` gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a name and authorizing the save operation or by canceling the save operation. `SFPPutFile` returns the user's input in a record of type `SFReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SFPPutFile` are

Trap macro	Selector
`_Pack3`	$0003

SPECIAL CONSIDERATIONS

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

Opening Files

You can use the `StandardGetFile` procedure to present the standard user interface when the user asks to open a file. If you need to add elements to the default dialog boxes or exercise greater control over user actions in the dialog box, use `CustomGetFile`.

If your application is designed to execute in system software versions earlier than version 7.0, you can use the corresponding procedures `SFGetFile` and `SFPGetFile`.

StandardGetFile

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

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

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

DESCRIPTION

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

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

The `fileFilter` parameter points to an optional file filter function, provided by your application, through which `StandardGetFile` passes files of the specified types. See "Writing a File Filter Function" on page 3-20 for a description of the file filter function.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `StandardGetFile` are

Trap macro	Selector
`_Pack3`	$0006

SPECIAL CONSIDERATIONS

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

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

CustomGetFile

Call the `CustomGetFile` procedure when your application requires more control over the Open dialog box than is possible using `StandardGetFile`.

```
PROCEDURE CustomGetFile (fileFilter: FileFilterYDProcPtr;
                           numTypes: Integer;
                           typeList: SFTypeList;
                           VAR reply: StandardFileReply;
                           dlgID: Integer;
                           where: Point;
                           dlgHook: DlgHookYDProcPtr;
                           filterProc: ModalFilterYDProcPtr;
                           activeList: Ptr;
                           activateProc: ActivateYDProcPtr;
                           yourDataPtr: UNIV Ptr);
```

`fileFilter`	A pointer to an optional file filter function, provided by your application, through which `CustomGetFile` passes files of the specified types.
`numTypes`	The number of file types to be displayed. If you specify a `numTypes` value of –1, the first filtering passes files of all types.
`typeList`	A list of file types to be displayed.
`reply`	The reply record, which `CustomGetFile` fills in before returning.
`dlgID`	The resource ID of a customized dialog template. To use the standard template, set this parameter to 0.
`where`	The upper-left corner of the dialog box in global coordinates. If you specify the point (–1,–1), `CustomGetFile` automatically centers the dialog box on the screen.
`dlgHook`	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you have not added any items to the dialog box and want the standard items handled in the standard ways. See "Writing a Dialog Hook Function" on page 3-21 for a description of the dialog hook function.
`filterProc`	A pointer to your modal-dialog filter function, which determines how `ModalDialog` filters events when called by `CustomGetFile`. Specify a value of `NIL` if you are not supplying your own function. See "Writing a Modal-Dialog Filter Function" on page 3-28 for a description of the modal-dialog filter function.
`activeList`	A pointer to a list of all dialog items that can be activated—that is, made the target of keyboard input. The list is stored as an array of 16-bit integers. The first integer is the number of items in the list. The remaining integers are the item numbers of all possible keyboard targets, in the order that they are activated by the Tab key. If you supply an `activeList` parameter of `NIL`, `CustomGetFile` directs all keyboard input to the displayed list.

`activateProc`
A pointer to your activation procedure, which controls the highlighting of dialog items that are defined by your application and that can receive keyboard input. See "Writing an Activation Procedure" on page 3-30 for a description of the activation procedure.

`yourDataPtr`
A pointer to optional data supplied by your application. When `CustomGetFile` calls any of your callback routines, it pushes this parameter on the stack, making the data available to your callback routines. If you are not supplying any data of your own, specify a value of `NIL`.

DESCRIPTION

The `CustomGetFile` procedure is an alternative to `StandardGetFile` when you want to use a customized dialog box or handle the default Open dialog box in a customized way. `CustomGetFile` presents a dialog box through which the user specifies the name and location of a file to be opened. While the dialog box is active, `CustomGetFile` gets and handles events until the user completes the interaction, either by selecting a file to open or by canceling the operation. `CustomGetFile` returns the user's input in a record of type `StandardFileReply`.

The first four parameters are similar to the same parameters in `StandardGetFile`. The `fileFilter`, `numTypes`, and `typeList` parameters determine which files appear in the list of choices. If you specify a value of –1 in the `numTypes` parameter, `CustomGetFile` displays or passes to your file filter function all files and folders (not just the files) at the current level of the display hierarchy. If you provide a filter function, `CustomGetFile` passes it both the pointer to the catalog entry for each file to be processed and also a pointer to the optional data passed by your application in its call to `CustomGetFile`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `CustomGetFile` are

Trap macro	Selector
`_Pack3`	$0008

SPECIAL CONSIDERATIONS

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

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

SFGetFile

Use the `SFGetFile` procedure to display the default Open dialog box when the user is opening a file.

```
PROCEDURE SFGetFile (where: Point; prompt: Str255;
                     fileFilter: FileFilterProcPtr;
                     numTypes: Integer; typeList: SFTypeList;
                     dlgHook: DlgHookProcPtr; VAR reply: SFReply);
```

`where`	The upper-left corner of the dialog box, in global coordinates.
`prompt`	Ignored.
`fileFilter`	A pointer to an optional file filter function, provided by your application, through which `SFGetFile` passes files of the specified types.
`numTypes`	The number of file types to be displayed. If you specify a `numTypes` value of –1, the first filtering passes files of all types.
`typeList`	A list of file types to be displayed.
`dlgHook`	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you want the standard items handled in the standard ways.
`reply`	The reply record, which `SFGetFile` fills in before returning.

DESCRIPTION

The `SFGetFile` procedure displays a dialog box listing the names of a specific group of files from which the user can select one to be opened (as during an Open menu command). During the dialog, `SFGetFile` gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a file to open or by canceling the open operation. `SFGetFile` returns the user's input in a record of type `SFReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SFGetFile` are

Trap macro	Selector
`_Pack3`	$0002

SPECIAL CONSIDERATIONS

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

SFPGetFile

Call the `SFPGetFile` procedure when your application requires more control over the Open dialog box than is possible using `SFGetFile`.

```
PROCEDURE SFPGetFile (where: Point; prompt: Str255;
                        fileFilter: FileFilterProcPtr;
                        numTypes: Integer; typeList: SFTypeList;
                        dlgHook: DlgHookProcPtr;
                        VAR reply: SFReply; dlgID: Integer;
                        filterProc: ModalFilterProcPtr);
```

`where`	The upper-left corner of the dialog box, in global coordinates.
`prompt`	Ignored.
`fileFilter`	A pointer to an optional file filter function, provided by your application, through which `SFPGetFile` passes files of the specified types.
`numTypes`	The number of file types to be displayed. If you specify a `numTypes` value of –1, the first filtering passes files of all types.
`typeList`	A list of file types to be displayed.
`dlgHook`	A pointer to your dialog hook function, which handles item selections received from the Dialog Manager. Specify a value of `NIL` if you have not added any items to the dialog box and want the standard items handled in the standard ways.
`reply`	The reply record, which `SFPGetFile` fills in before returning.
`dlgID`	The resource ID of a customized dialog template.
`filterProc`	A pointer to your modal-dialog filter function, which determines how the `ModalDialog` procedure filters events when called by the `SFPGetFile` procedure. Specify a value of `NIL` if you are not supplying your own function.

DESCRIPTION

The `SFPGetFile` procedure is an alternative to `SFGetFile` when you want to display a customized Open dialog box or handle the default dialog box in a customized way. During the dialog, `SFPGetFile` gets and handles events (possibly with the assistance of application-defined callback routines) until the user completes the interaction, either by selecting a file to open or by canceling the open operation. `SFPGetFile` returns the user's input in a record of type `SFReply`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for `SFPGetFile` are

Trap macro	Selector
`_Pack3`	$0004

SPECIAL CONSIDERATIONS

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

Application-Defined Routines

This section describes the application-defined routines whose addresses you pass to some of the Standard File Package routines. You can define

- a file filter function for determining which files the user can open
- a dialog hook function for handling user actions in the dialog boxes
- a modal-dialog filter function for handling user events received from the Event Manager
- an activation procedure for highlighting the display when keyboard input is directed at a customized field defined by your application

File Filter Functions

You specify a file filter function to determine which files appear in the displayed list of files and folders when the user is opening a file. You can define a standard or custom file filter.

MyStandardFileFilter

A file filter function whose address is passed to `StandardGetFile` should have the following form:

```
FUNCTION MyStandardFileFilter (pb: CInfoPBPtr): Boolean;
```

`pb`	A pointer to a catalog information parameter block.

DESCRIPTION

When `StandardGetFile` is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application's specifications. If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

When your file filter function is called, it is passed, in the `pb` parameter, a pointer to a catalog information parameter block. See the chapter "File Manager" in this book for a description of the fields of this parameter block.

Your function evaluates the catalog information parameter block and returns a Boolean value that determines whether the file is filtered (that is, a value of `TRUE` suppresses display of the filename, and a value of `FALSE` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

SEE ALSO

See "Writing a File Filter Function" on page 3-20 for a sample file filter function.

MyCustomFileFilter

A file filter function whose address is passed to `CustomGetFile` should have the following form:

```
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr):
                                   Boolean;
```

`pb`	A pointer to a catalog information parameter block.
`myDataPtr`	A pointer to the optional data whose address is passed to `CustomGetFile`.

DESCRIPTION

When `CustomGetFile` is displaying the contents of a volume or folder, it checks the file type of each file and filters out files whose types do not match your application's specifications. If your application also supplies a file filter function, the Standard File Package calls that function each time it identifies a file of an acceptable type.

When your file filter function is called, it is passed, in the `pb` parameter, a pointer to a catalog information parameter block. See the chapter "File Manager" in this book for a description of the fields of this parameter block.

Your function evaluates the catalog information parameter block and returns a Boolean value that determines whether the file is filtered (that is, a value of `TRUE` suppresses display of the filename, and a value of `FALSE` allows the display). If you do not supply a file filter function, the Standard File Package displays all files of the specified types.

SEE ALSO

See "Writing a File Filter Function" on page 3-20 for a sample file filter function.

Dialog Hook Functions

A dialog hook function handles user selections in a dialog box.

MyDlgHook

A dialog hook function should have the following form:

```
FUNCTION MyDlgHook (item: Integer; theDialog: DialogPtr;
                      myDataPtr: Ptr): Integer;
```

`item`	The number of the item selected.
`theDialog`	A pointer to the dialog record of the dialog box.
`myDataPtr`	A pointer to the optional data whose address is passed to `CustomGetFile` or `CustomPutFile`.

DESCRIPTION

You supply a dialog hook function to handle user selections of items that you added to a dialog box. If you provide a dialog hook function, `CustomPutFile` and `CustomGetFile` call your function immediately after calling `ModalDialog`. They pass your function the item number returned by `ModalDialog`, a pointer to the dialog record, and a pointer to the data received from your application, if any.

Your dialog hook function returns as its function result an integer that is either the item number passed to it or some other item number. If your dialog hook function does not handle a selection, it should pass the item number back to the Standard File Package for processing by setting its return value equal to the item number. If your dialog hook function does handle the selection, it should pass back `sfHookNullEvent` or the number of some other pseudo-item.

SEE ALSO

See "Writing a Dialog Hook Function" on page 3-21 for a sample dialog hook function.

Modal-Dialog Filter Functions

A modal-dialog filter function controls events closer to their source by filtering the events received from the Event Manager. The Standard File Package itself contains an internal modal-dialog filter function that maps keypresses and other user input onto the equivalent dialog box items. If you also want to process events at this level, you can supply your own filter function.

MyModalFilter

A modal-dialog filter function whose address is passed to `SFPGetFile` or `SFPPutFile` should have the following form:

```
FUNCTION MyModalFilter (theDialog: DialogPtr;
                        VAR theEvent: EventRecord;
                        VAR itemHit: Integer): Boolean;
```

`theDialog`	A pointer to the dialog record of the dialog box.
`theEvent`	The event record for the event.
`itemHit`	The number of the item selected.

DESCRIPTION

Your modal-dialog filter function determines how the Dialog Manager procedure `ModalDialog` filters events. The `ModalDialog` procedure retrieves events by calling the Event Manager function `GetNextEvent`. The Standard File Package contains an internal filter function that performs some preliminary processing on each event it receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `FALSE`, `ModalDialog` processes the event through its own filters. If your function returns a value of `TRUE`, `ModalDialog` returns with no further action.

SEE ALSO

See "Writing a Modal-Dialog Filter Function" on page 3-28 for a sample modal-dialog filter function.

MyModalFilterYD

A modal-dialog filter function whose address is passed to `CustomGetFile` or `CustomPutFile` should have the following form:

```
FUNCTION MyModalFilterYD (theDialog: DialogPtr;
                          VAR theEvent: EventRecord;
                          VAR itemHit: Integer;
                          myDataPtr: Ptr): Boolean;
```

`theDialog`	A pointer to the dialog record of the dialog box.
`theEvent`	The event record for the event.
`itemHit`	The number of the item selected.
`myDataPtr`	A pointer to the optional data whose address is passed to `CustomGetFile` or `CustomPutFile`.

DESCRIPTION

Your modal-dialog filter function determines how the Dialog Manager procedure `ModalDialog` filters events. The `ModalDialog` procedure retrieves events by calling the Event Manager function `GetNextEvent`. The Standard File Package contains an internal filter function that performs some preliminary processing on each event it receives. If you provide a modal-dialog filter function, `ModalDialog` calls your filter function after it calls the internal Standard File Package filter function and before it sends the event to your dialog hook function.

Your modal-dialog filter function returns a Boolean value that reports whether it handled the event. If your function returns a value of `FALSE`, `ModalDialog` processes the event through its own filters. If your function returns a value of `TRUE`, `ModalDialog` returns with no further action.

SEE ALSO

See “Writing a Modal-Dialog Filter Function” on page 3-28 for a sample modal-dialog filter function.

Activation Procedures

An activation procedure controls the highlighting of dialog items that are defined by your application and can receive keyboard input.

MyActivateProc

An activation procedure should have the following form:

```
PROCEDURE MyActivateProc (theDialog: DialogPtr; itemNo: Integer;
                           activating: Boolean; myDataPtr: Ptr);
```

`theDialog`	A pointer to the dialog record of the dialog box.
`itemNo`	The number of the item selected.
`activating`	A Boolean value that specifies whether the field is being activated (`TRUE`) or deactivated (`FALSE`).
`myDataPtr`	A pointer to the optional data whose address is passed to `CustomGetFile` or `CustomPutFile`.

DESCRIPTION

Your activation procedure controls the highlighting of dialog items that are defined by your application and can receive keyboard input. Ordinarily, you need to supply an activation procedure only if your application builds a list from which the user can select entries. The Standard File Package supplies the activation procedure for the file display list and for all TextEdit fields. You can also use the activation procedure to keep track of which field is receiving keyboard input, if your application needs that information.

Your application is responsible for removing the highlighting when one of its fields becomes inactive and for adding the highlighting when one of its fields becomes active. The Standard File Package can handle the highlighting of all TextEdit fields, even those defined by your application.

Summary of the Standard File Package

Pascal Summary

Constants

```
CONST
   {Gestalt selector and reply}
   gestaltStandardFileAttr =  'stdf';
   gestaltStandardFile58   =  0;

   {standard dialog resource IDs}
   sfPutDialogID           =  -6043;   {Save dialog box}
   sfGetDialogID           =  -6042;   {Open dialog box}

   {items that appear in both the Open and Save dialog boxes}
   sfItemOpenButton        =  1;       {Save or Open button}
   sfItemCancelButton      =  2;       {Cancel button}
   sfItemBalloonHelp       =  3;       {Balloon Help}
   sfItemVolumeUser        =  4;       {volume icon and name}
   sfItemEjectButton       =  5;       {Eject button}
   sfItemDesktopButton     =  6;       {Desktop button}
   sfItemFileListUser      =  7;       {display list}
   sfItemPopUpMenuUser     =  8;       {directory pop-up menu}
   sfItemDividerLinePict   =  9;       {dividing line between buttons}

   {items that appear in Save dialog boxes only}
   sfItemFileNameTextEdit  =  10;      {filename field}
   sfItemPromptStaticText  =  11;      {filename prompt text area}
   sfItemNewFolderUser     =  12;      {New Folder button}

   {pseudo-items available prior to version 7.0}
   sfHookFirstCall         =  -1;      {initialize display}
   sfHookCharOffset        =  $1000;   {offset for character input}
   sfHookNullEvent         =  100;     {null event}
   sfHookRebuildList       =  101;     {redisplay list}
   sfHookFolderPopUp       =  102;     {display parent-directory menu}
   sfHookOpenFolder        =  103;     {display contents of selected }
                                       { folder or volume}
```

```
{additional pseudo-items introduced in version 7.0}
sfHookLastCall          =  -2;      {clean up after display}
sfHookOpenAlias         =  104;     {resolve alias}
sfHookGoToDesktop       =  105;     {display contents of desktop}
sfHookGoToAliasTarget   =  106;     {select target of alias}
sfHookGoToParent        =  107;     {display contents of parent}
sfHookGoToNextDrive     =  108;     {display contents of next drive}
sfHookGoToPrevDrive     =  109;     {display contents of previous drive}
sfHookChangeSelection   =  110;     {select target of reply record}
sfHookSetActiveOffset   =  200;     {switch active item}

{refCon field in the window record in the dialog record}
sfMainDialogRefCon      =  'stdf';  {main dialog box}
sfNewFolderDialogRefCon =  'nfdr';  {New Folder dialog box}
sfReplaceDialogRefCon   =  'rplc';  {name conflict dialog box}
sfStatWarnDialogRefCon  =  'stat';  {stationery warning}
sfErrorDialogRefCon     =  'err ';  {general error report}
sfLockWarnDialogRefCon  =  'lock';  {software lock warning}

{resource IDs and item numbers of pre-7.0 dialog boxes}
putDlgID                =  -3999;   {Save dialog box}
putSave                 =  1;       {Save button}
putCancel               =  2;       {Cancel button}
putEject                =  5;       {Eject button}
putDrive                =  6;       {Drive button}
putName                 =  7;       {filename field}

getDlgID                =  -4000;   {Open dialog box}
getOpen                 =  1;       {Open button}
getCancel               =  3;       {Cancel button}
getEject                =  5;       {Eject button}
getDrive                =  6;       {Drive button}
getNmList               =  7;       {list of names}
getScroll               =  8;       {scroll bar}
```

Data Types

Standard File Reply Records

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

   SFReply           =           {original standard file reply record}
   RECORD
      good:          Boolean;    {TRUE if user did not cancel}
      copy:          Boolean;    {reserved}
      fType:         OSType;     {file type}
      vRefNum:       Integer;    {working directory reference number}
      version:       Integer;    {reserved}
      fName:         Str63;      {filename}
   END;
```

Standard File Type List

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

Callback Routine Pointer Types

```
   DlgHookProcPtr       =  ProcPtr;    {dialog hook function}
   DlgHookYDProcPtr     =  ProcPtr;    {dialog hook function with data}
   FileFilterProcPtr    =  ProcPtr;    {file filter function}
   FileFilterYDProcPtr  =  ProcPtr;    {file filter function with data}
   ModalFilterProcPtr   =  ProcPtr;    {modal-dialog filter}
   ModalFilterYDProcPtr =  ProcPtr;    {modal-dialog filter with data}
   ActivateYDProcPtr    =  ProcPtr;    {activation procedure}
```

Standard File Package Routines

Saving Files

```
PROCEDURE StandardPutFile   (prompt: Str255; defaultName: Str255;
                             VAR reply: StandardFileReply);
PROCEDURE CustomPutFile     (prompt: Str255; defaultName: Str255;
                             VAR reply: StandardFileReply; dlgID: Integer;
                             where: Point; dlgHook: DlgHookYDProcPtr;
                             filterProc: ModalFilterYDProcPtr;
                             activeList: Ptr;
                             activateProc: ActivateYDProcPtr;
                             yourDataPtr: UNIV Ptr);
PROCEDURE SFPutFile         (where: Point; prompt: Str255;
                             origName: Str255; dlgHook: DlgHookProcPtr;
                             VAR reply: SFReply);
PROCEDURE SFPPutFile        (where: Point; prompt: Str255;
                             origName: Str255; dlgHook: DlgHookProcPtr;
                             VAR reply: SFReply; dlgID: Integer;
                             filterProc: ModalFilterProcPtr);
```

Opening Files

```
PROCEDURE StandardGetFile   (fileFilter: FileFilterProcPtr;
                             numTypes: Integer; typeList: SFTypeList;
                             VAR reply: StandardFileReply);
PROCEDURE CustomGetFile     (fileFilter: FileFilterYDProcPtr;
                             numTypes: Integer; typeList: SFTypeList;
                             VAR reply: StandardFileReply; dlgID: Integer;
                             where: Point; dlgHook: DlgHookYDProcPtr;
                             filterProc: ModalFilterYDProcPtr;
                             activeList: Ptr;
                             activateProc: ActivateYDProcPtr;
                             yourDataPtr: UNIV Ptr);
PROCEDURE SFGetFile         (where: Point; prompt: Str255;
                             fileFilter: FileFilterProcPtr;
                             numTypes: Integer; typeList: SFTypeList;
                             dlgHook: DlgHookProcPtr; VAR reply: SFReply);
PROCEDURE SFPGetFile        (where: Point; prompt: Str255;
                             fileFilter: FileFilterProcPtr;
                             numTypes: Integer; typeList: SFTypeList;
                             dlgHook: DlgHookProcPtr; VAR reply: SFReply;
                             dlgID: Integer;
                             filterProc: ModalFilterProcPtr);
```

Application-Defined Routines

```
FUNCTION MyStandardFileFilter
                            (pb: CInfoPBPtr): Boolean;
FUNCTION MyCustomFileFilter (pb: CInfoPBPtr; myDataPtr: Ptr): Boolean;
FUNCTION MyDlgHook          (item: Integer; theDialog: DialogPtr;
                             myDataPtr: Ptr): Integer;
FUNCTION MyModalFilter      (theDialog: DialogPtr;
                             VAR theEvent: EventRecord;
                             VAR itemHit: Integer): Boolean;
FUNCTION MyModalFilterYD    (theDialog: DialogPtr;
                             VAR theEvent: EventRecord;
                             VAR itemHit: Integer; myDataPtr: Ptr): Boolean;
PROCEDURE MyActivateProc    (theDialog: DialogPtr; itemNo: Integer;
                             activating: Boolean; myDataPtr: Ptr);
```

C Summary

Constants

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

/*standard dialog resource IDs*/
enum {sfPutDialogID           = (-6043)}; /*Save dialog box*/
enum {sfGetDialogID           = (-6042)}; /*Open dialog box*/

/*items that appear in both the Open and Save dialog boxes/*
enum {sfItemOpenButton        = 1};    /*Save or Open button*/
enum {sfItemCancelButton      = 2};    /*Cancel button*/
enum {sfItemBalloonHelp       = 3};    /*Balloon Help*/
enum {sfItemVolumeUser        = 4};    /*volume icon and name*/
enum {sfItemEjectButton       = 5};    /*Eject button*/
enum {sfItemDesktopButton     = 6};    /*Desktop button*/
enum {sfItemFileListUser      = 7};    /*display list*/
enum {sfItemPopUpMenuUser     = 8};    /*directory pop-up menu*/
enum {sfItemDividerLinePict   = 9};    /*dividing line between buttons*/

/*items that appear in Save dialog boxes only*/
enum {sfItemFileNameTextEdit  = 10};   /*filename field*/
enum {sfItemPromptStaticText  = 11};   /*filename prompt text area*/
enum {sfItemNewFolderUser     = 12};   /*New Folder button*/
```

```
/*pseudo-items available prior to version 7.0*/
enum {sfHookFirstCall         = (-1)}; /*initialize display*/
enum {sfHookCharOffset        = 0x1000};/*offset for character input*/
enum {sfHookNullEvent         = 100};  /*null event*/
enum {sfHookRebuildList       = 101};  /*redisplay list*/
enum {sfHookFolderPopUp       = 102};  /*display parent-directory menu*/
enum {sfHookOpenFolder        = 103};  /*display contents of selected */
                                       /* folder or volume*/

/*additional pseudo-items introduced in version 7.0*/
enum {sfHookLastCall          = (-2)}; /*clean up after display*/
enum {sfHookOpenAlias         = 104};  /*resolve alias*/
enum {sfHookGoToDesktop       = 105};  /*display contents of desktop*/
enum {sfHookGoToAliasTarget   = 106};  /*select target of alias*/
enum {sfHookGoToParent        = 107};  /*display contents of parent*/
enum {sfHookGoToNextDrive     = 108};  /*display contents of next drive*/
enum {sfHookGoToPrevDrive     = 109};  /*display contents of previous drive*/
enum {sfHookChangeSelection   = 110};  /*select target of reply record*/
enum {sfHookSetActiveOffset   = 200};  /*switch active item*/

/*refCon field in the window record in the dialog record*/
#define sfMainDialogRefCon       'stdf'   /*main dialog box*/
#define sfNewFolderDialogRefCon  'nfdr'   /*New Folder dialog box*/
#define sfReplaceDialogRefCon    'rplc'   /*name conflict dialog box*/
#define sfStatWarnDialogRefCon   'stat'   /*stationery warning*/
#define sfErrorDialogRefCon      'err '   /*general error report*/
#define sfLockWarnDialogRefCon   'lock'   /*software lock warning*/

/*resource IDs and item numbers of pre-7.0 dialog boxes*/
enum {putDlgID                = -3999};   /*Save dialog box*/
enum {putSave                 = 1};       /*Save button*/
enum {putCancel               = 2};       /*Cancel button*/
enum {putEject                = 5};       /*Eject button*/
enum {putDrive                = 6};       /*Drive button*/
enum {putName                 = 7};       /*filename field*/

enum {getDlgID                = -4000};   /*Open dialog box*/
enum {getOpen                 = 1};       /*Open button*/
enum {getCancel               = 3};       /*Cancel button*/
enum {getEject                = 5};       /*Eject button*/
enum {getDrive                = 6};       /*Drive button*/
enum {getNmList               = 7};       /*list of names*/
enum {getScroll               = 8};       /*scroll bar*/
```

Data Types

Standard File Reply Records

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

typedef struct StandardFileReply StandardFileReply;


struct SFReply {                  /*original standard file reply record*/
      Boolean        good;        /*TRUE if user did not cancel*/
      Boolean        copy;        /*reserved*/
      OSType         fType;       /*file type*/
      short          vRefNum;     /*working directory reference number*/
      short          version;     /*reserved*/
      Str63          fName;       /*filename*/
};

typedef struct SFReply SFReply;
```

Standard File Type List

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

Callback Routine Pointer Types

```
/*dialog hook function*/
typedef pascal short (*DlgHookProcPtr)
                             (short item, DialogPtr theDialog);
/*dialog hook function with data*/
typedef pascal short (*DlgHookYDProcPtr)
                             (short item, DialogPtr theDialog,
                              void *yourDataPtr);
```

```
/*file filter function*/
typedef pascal Boolean (*FileFilterProcPtr)
                            (ParmBlkPtr PB);
/*file filter function with data*/
typedef pascal Boolean (*FileFilterYDProcPtr)
                            (ParmBlkPtr PB, void *yourDataPtr);
/*modal-dialog filter*/
typedef pascal ProcPtr ModalFilterProcPtr;
                            (DialogPtr theDialog, EventRecord *theEvent,
                             short *itemHit);
/*modal-dialog filter with data*/
typedef pascal Boolean (*ModalFilterYDProcPtr)
                            (DialogPtr theDialog, EventRecord *theEvent,
                             short *itemHit, void *yourDataPtr);
/*activation procedure*/
typedef pascal void (*ActivateYDProcPtr)
                            (DialogPtr theDialog,
                             short itemNo, Boolean activating,
                             void *yourDataPtr);
```

Standard File Package Routines

Saving Files

```
pascal void StandardPutFile (const Str255 prompt, const Str255 defaultName,
                             StandardFileReply *reply);
pascal void CustomPutFile   (const Str255 prompt, const Str255 defaultName,
                             StandardFileReply *reply, short dlgID,
                             Point where, DlgHookYDProcPtr dlgHook,
                             ModalFilterYDProcPtr filterProc,
                             short *activeList,
                             ActivateYDProcPtr activateProc,
                             void *yourDataPtr);
pascal void SFPutFile       (Point where, const Str255 prompt,
                             const Str255 origName, DlgHookProcPtr dlgHook,
                             SFReply *reply);
pascal void SFPPutFile      (Point where, const Str255 prompt,
                             const Str255 origName, DlgHookProcPtr dlgHook,
                             SFReply *reply, short dlgID,
                             ModalFilterProcPtr filterProc);
```

Opening Files

```
pascal void StandardGetFile (const Str255 prompt,
                             FileFilterProcPtr fileFilter,
                             short numTypes, SFTypeList typeList,
                             StandardFileReply *reply);
pascal void CustomGetFile   (FileFilterYDProcPtr fileFilter,
                             short numTypes, SFTypeList typeList,
                             StandardFileReply *reply, short dlgID,
                             Point where, DlgHookYDProcPtr dlgHook,
                             ModalFilterYDProcPtr filterProc,
                             short *activeList,
                             ActivateYDProcPtr activateProc,
                             void *yourDataPtr);
pascal void SFGetFile       (Point where, const Str255 prompt,
                             FileFilterProcPtr fileFilter, short numTypes,
                             SFTypeList typeList, DlgHookProcPtr dlgHook,
                             SFReply *reply);
pascal void SFPGetFile      (Point where, const Str255 prompt,
                             FileFilterProcPtr fileFilter,
                             short numTypes, SFTypeList typeList,
                             DlgHookProcPtr dlgHook, SFReply *reply,
                             short dlgID, ModalFilterProcPtr filterProc);
```

Application-Defined Routines

```
pascal Boolean MyStandardFileFilter
                            (CInfoPBPtr pb);
pascal Boolean MyCustomFileFilter
                            (CInfoPBPtr pb, Ptr myDataPtr);
pascal short MyDlgHook      (short item, DialogPtr theDialog,
                             Ptr myDataPtr);
pascal Boolean MyModalFilter(DialogPtr theDialog,
                             EventRecord *theEvent, short *itemHit);
pascal Boolean MyModalFilterYD
                            (DialogPtr theDialog,
                             EventRecord *theEvent, short *itemHit,
                             Ptr myDataPtr);
pascal void MyActivateProc  (DialogPtr theDialog, short itemNo,
                             Boolean activating, Ptr myDataPtr);
```

Assembly-Language Summary

Data Structures

New Standard File Reply Record

0	`sfGood`	byte	command-valid flag
1	`sfReplacing`	byte	replace existing file flag
2	`sfType`	long	file type
6	`sfFile`	70 bytes	selected item
76	`sfScript`	word	display script
78	`sfFlags`	word	Finder flags from catalog
80	`sfIsFolder`	byte	folder flag
81	`sfIsVolume`	byte	volume flag
82	`sfReserved1`	long	reserved
86	`sfReserved2`	word	reserved

Old Standard File Reply Record

0	`good`	byte	command-valid flag
1	`copy`	byte	reserved
2	`fType`	long	file type
6	`vRefNum`	word	working directory reference number
8	`version`	word	reserved
10	`fName`	64 bytes	name of file (length byte followed by up to 63 characters)

Trap Macros

Trap Macro Requiring Routine Selector

`_Pack3`

Selector	Routine
$0001	`SFPutFile`
$0002	`SFGetFile`
$0003	`SFPPutFile`
$0004	`SFPGetFile`
$0005	`StandardPutFile`
$0006	`StandardGetFile`
$0007	`CustomPutFile`
$0008	`CustomGetFile`

Global Variables

`CurDirStore`	long	The directory ID of the current directory.
`SFSaveDisk`	word	The negative of the volume reference number of the current volume.