Customizing Your Interface Special-purpose dialog boxes

If your application requires it, you can customize the user interface for identifying files.

Note: Alter the dialog boxes only if necessary. Apple, does not guarantee future compatibility if you use a customized dialog box.

To customize a dialog box, you should

- design your display 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

Whether or not you change the dialog box display, you can write your own dialog hook callback function to handle user actions in the dialog box.

Customized Dialog Boxes

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

The following code example shows the resource definition of the default **Open** dialog box, in Rez input format. (Files in Rez format may be used with SARez and SADeRez, provided with Think C.)

The definition of the default **Open** dialog box

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

The next code example shows the resource definition of the default **Save** dialog box, in Rez input format.

The definition of the default **Save** dialog box

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

The default **Standard File Package** dialog boxes now 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 grafPorts for the default dialog boxes. (See the

Dialog Manager for a description of the 'dctb' resource.) If you create your own dialog boxes, include 'dctb' 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 nine items in a customized dialog box to have the same functions as the corresponding items in the **StandardGetFile** dialog box; **CustomPutFile** expects the first twelve 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.

The next code example shows the dialog item list for the default **Open** dialog box, in Rez input format. The constant statements in the next section, **Callback Routines**, list which elements the items represent in the dialog boxes.

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 },
    }       };
```

The dialog item list for the default **Save** dialog box, in Rez

```
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. HelpItem specifies the resource ID of the 'hdlg' resource that contains the help strings for the standard dialog items. 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. For more information about **Balloon Help**, see the **Help Manager** .

Callback Routines

You can supply callback routines that control these elements of the user interface:

- determining which files the user can open
- handling user actions in the dialog boxes
- handling user events received from the **Event Manager**
- highlighting the display when keyboard input is directed at a customized field defined by your application

You can also supply data of your own to be passed into the callback routines through a new parameter, yourDataPtr, that you can pass to **CustomGetFile** and **CustomPutFile**.

A file filter function determines which files appear in the display 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**. 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 **File Manager** .It 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). The **Standard File Package** displays all files of the specified types, if you do not supply a file filter function of your own.

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

```
pascal Boolean MyStandardFileFilter (ParmBlkPtr pb);
```

When your file filter function is called by **CustomGetFile**, 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:

```
pascal Boolean MyCustomFileFilter (ParmBlkPtr pb, void *myDataPtr );
```

A **dialog hook function** handles item hits in the dialog box. It receives a dialog record and an item number from the **ModalDialog** procedure by means of the **Standard File Package** each time the user causes a hit on one of the dialog items. Your dialog hook function checks the item number of each item hit, and then either handles the hit 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**. It passes 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:

pascal short MyDlgHook (short item, DialogPtr theDialog, void *myDataPtr);

Your dialog hook function returns an item number or the sfHookNullItem constant as its function result. If it returns one of the item numbers in the following list of constants, the **Standard File Package** handles the item hit as described. If your dialog hook function does not handle an item hit, it should pass the item number back to the **Standard File Package** for processing by setting its return value equal to the item number.

When your application handles the item hit, it should return the sfHookNullEvent constant. When the **Standard File Package** receives either sfHookNullEvent or an item number that it doesn't recognize from a dialog hook function, it does nothing.

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

The **Standard File Package** uses a set of modal-dialog filter functions (described later in this section) to map user actions during the dialog onto the defined item numbers. Some of the mapping is indirect. A click on 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 hit onto the pseudo-item sfHookOpenFolder.

The lists that follow summarize when various items are generated and how they are handled. The lists describe the simplest mouse action that generates each item; many of the items can also be generated by keyboard actions, as described in **Presenting the Default Interface**.

The first twelve defined constants represent the items in the **Save** and **Open** dialog boxes. 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. Except under extraordinary circumstances, your dialog hook function always passes any of the first twelve item numbers back to the **Standard File Package** for processing.

Constant	**Cause**	**Effect**
sfItemOpenButton	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	The user clicks **Cancel**.	The **Standard File Package** sets sfGood to FALSE, removes the dialog box, and returns.
sfItemVolumeUser	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	The user clicks **Eject**.	The **Standard File Package** ejects the volume that is currently selected.
sfItemDesktopButton	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 in a list later in this section.	The **Standard File Package** displays the contents of the next drive.
sfItemFileListUser	The user clicks an item in the display list. The **Standard File Package** updates the selection and generates this item for your information.	No action.
sfItemPopUpMenuUser	Never generated. The **Standard File Package**'s modal-dialog filter function maps clicks on the directory pop-up menu to sfHookFolderPopUp, described in a list later in this section.	No action.
sfItemFileNameTextEdit	The user clicks in the filename fieldTextEdit and the **Standard File Package** process mouse clicks in the filename field, but the item number is generated for your information.	No action.
sfItemNewFolderUser	The user clicks **New Folder**.	The **Standard FilePackage** 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	Cause	Response
sfHookFirstCall	The **Standard File Package** generates this item 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 it when you receive this item. You can specify where inthe file system the dialog box should open eitherby returning sfHookGoToDesktop or by changing the reply record and returning sfHookChangeSelection.
sfHookLastCall	The **Standard File Package** generates this item number 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	The **Standard File Package** issues this null item periodically if no user action has taken place.	Your application can use this 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	Cause	Effect
sfHookRebuildList	Your dialog hook function returns this item 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	Your application returns this value to the **Standard File Package** after changing the reply record so that it describes a different file or folder.	The **Standard File Package** rebuilds the display list to show the contents of the folder or volume containing the

		object describedin 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 particular item numbers back to the **Standard File Package** for processing.

Constant	**Cause**	**Effect**
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 displaylist.
sfHookFolderPopUp	The user clicks the directory pop-up menu.	The **Standard File Package** displays the pop-up menu showing all parent directories.
sfHookOpenFolder	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	The **Standard File Package** generates this item number 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	The user clicks the **Desktop** button.	The **Standard File Package** displays the contents of the desktop in the display list.
sfHookGoToAliasTarget	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	The user presses **Command-Up Arrow**.	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	The user presses **Command-Right Arrow**.	The **Standard File Package** displays the contents of the next volume.
sfHookGoToPrevDrive	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 hits 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 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 refCon field to determine the target of the dialog record.

The defined values for the refCon field represent the standard file dialog boxes.

Constant	Value	Dialog box
sfMainDialogRefCon	'stdf'	Main dialog box, either **Open** or **Save**
sfNewFolderDialogRefCon	'nfdr'	**New Folder** dialog box
sfReplaceDialogRefCon	'rplc'	Verification for replacing a file of the same name
sfStatWarnDialogRefCon	'stat'	Warning that the user is opening the master copy of a stationery pad, not a piece of stationery
sfErrorDialogRefCon	'err '	Report of a general error
sfLockWarnDialogRefCon	'lock'	Warning that the user is opening a locked file and will not be able to save any changes

A modal-dialog filter function controls events closer to their source by filtering the events received from the **Event Manager** . The modal-dialog filter function is described in the **Dialog Manager** . By itself, the **Standard File Package** contains a modal-dialog filter function that maps keypresses and other user input onto the equivalent dialog box item hits. If you want to process events yourself, you can supply your own filter function.

Your modal-dialog filter function determines how the **Dialog Manager** procedure **ModalDialog** filters events when called by the **CustomGetFile** and **CustomPutFile** procedures. (Those procedures retrieve item hits by calling **ModalDialog**.) **ModalDialog** retrieves events by calling the **Event Manager** function **GetNextEvent**. If you provide a modal-dialog filter function, **ModalDialog** calls your filter function before processing an event and passes it a pointer to the dialog record, a pointer to the event record, the item number, and a pointer to the data received from your application, if any.

pascal Boolean MyModalFilter (DialogPtr theDialog, EventRecord *theEvent, short *itemHit, void myDataPtr);

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.

This function is the same as the modal-dialog filter function passed directly to **ModalDialog** (described in the **Dialog Manager**, with the addition of the optional pointer to your own data.

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 earlier in the description of dialog hook functions.

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 that specifies whether the field is being activated (TRUE) or deactivated (FALSE), and a pointer to your own data.

pascal void MyActivateProc (DialogPtr theDialog, short itemNo, Boolean activating void *myDataPtr);

Compatibility With Earlier Procedures

The **Standard File Package** still recognizes all procedures available before system 7.0 (**SFGetFile** , **SFPutFile** , **SFPGetFile** , and **SFPPutFile**). It displays the new interface for all applications that use the default dialog boxes (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 new interface 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 (for example, by specifying co-ordinates 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.