

# 16B

## **MORE ON FILES — NAVIGATION SERVICES**

*Includes Demonstration Program Files<sup>2</sup>*

### **Introduction**

Navigation Services, which was introduced with Mac OS 8.5<sup>1</sup> as an alternative to, and ultimately as a replacement for, the Standard File Package described at Chapter 16, provides greatly enhanced functionality in the area of document management. Navigation Services provides for opening and saving documents, confirming saves and discarding changes, choosing a volume, folder, file, or file object, creating a new folder, file format translation, and easier navigation.

The dialog and alert boxes created by Navigation Services are all Appearance-compliant. Provided that your application provides an event-handling callback function, the primary dialog boxes are resizable and movable and the alert boxes are movable.

As is the case with the Standard File Package dialogs, the primary Navigation Services dialog boxes may be customised.

### **Navigation Services Dialog and Alert Boxes**

The primary dialog boxes created by Navigation Services are as follows:

- Open.
- Save.
- Choose a folder.
- Choose a volume.
- Choose a file.
- Choose a file object.

---

<sup>1</sup> Although introduced with Mac OS 8.5, Navigation Services may be used with Mac OS 7.5.5 or later provided Appearance Manager 1.0.1 or later is present. QuickTime is also required if previews of graphic documents are to be created and viewed. Macintosh Easy Open is required for document translation and the correct display of file types. On 680x0 systems, Navigation Services requires the CFM-68K Runtime Enabler.

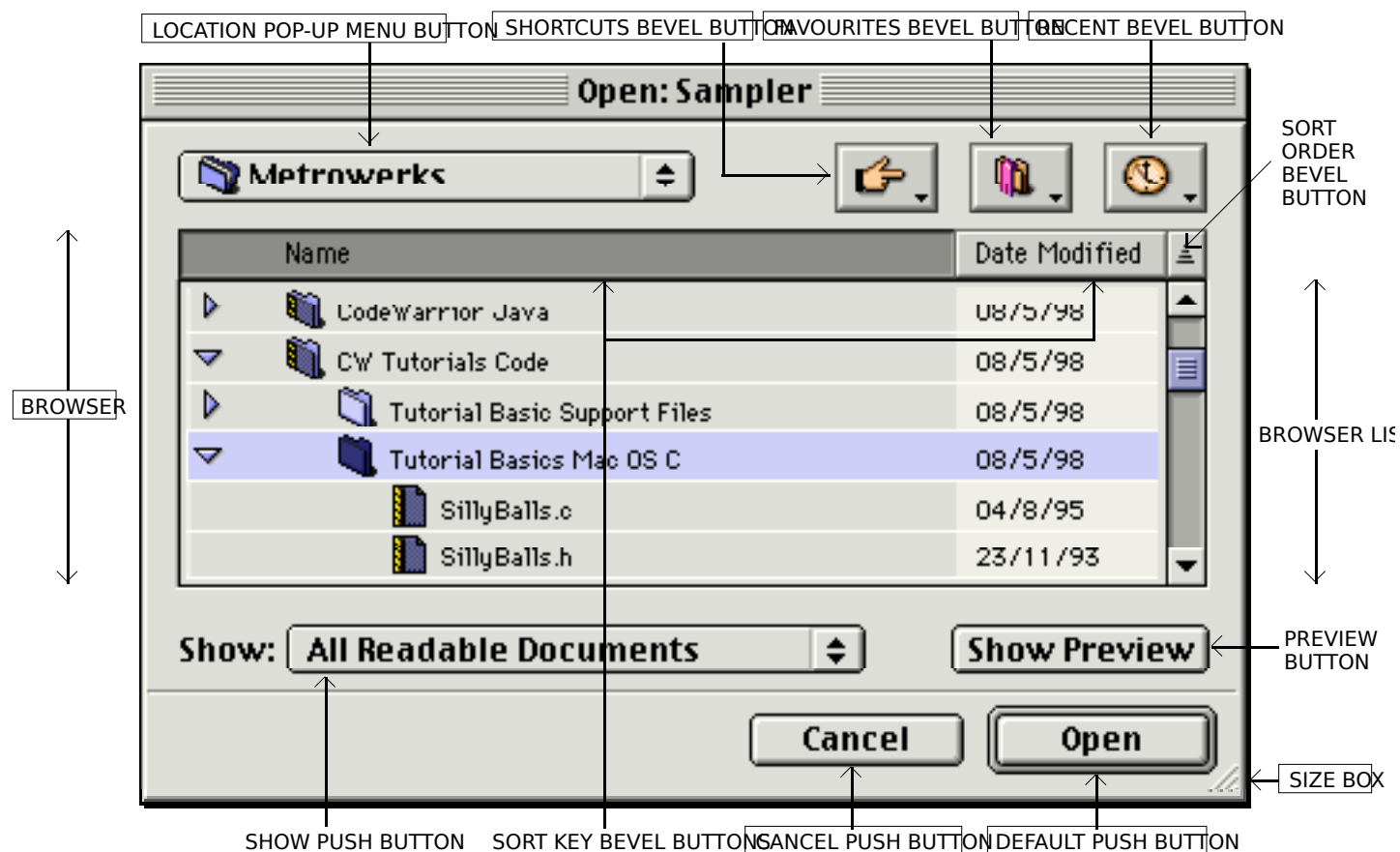
- Create a new folder.

The alert boxes created by Navigation Services are as follows:

- Save changes.
- Discard changes.

## Standard User Interface Elements in Dialog Boxes

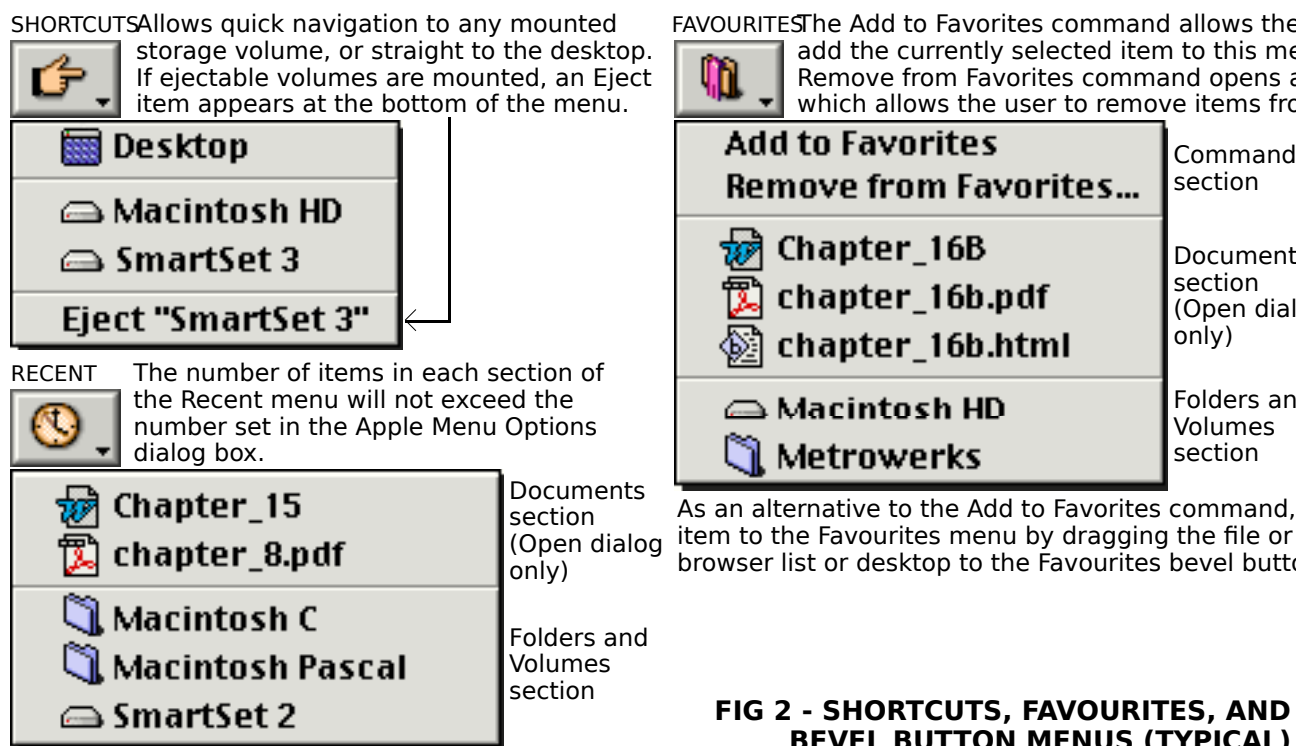
The standard user interface elements in Navigation Services primary dialog boxes are shown at Fig 1.



Note: The Open dialog box is shown. The names of the user interface elements common to all primary Navigation

**FIG 1 - STANDARD USER INTERFACE ELEMENTS IN NAVIGATION SERVICES DIALOG**

The menus associated with the standard user interface elements known as the Shortcut, Favourites, and Recent bevel buttons are shown in more detail at Fig 2.



**FIG 2 - SHORTCUTS, FAVOURITES, AND BEVEL BUTTON MENUS (TYPICAL)**

## ***Moving and Resizing Navigation Services Dialog Boxes***

Navigation Services primary dialog boxes are movable and resizable only if your application provides an application-defined event-handling callback function to handle update events. A universal procedure pointer to your event-handling function is passed in the `eventProc` parameter of Navigation Services functions such as `NavGetFile` (see below).

## ***Browser List Expansion***

When the user resizes the dialog box using the size box, the browser list expands proportionately. The date format in the browser list changes as the browser list expands.

## ***Sort Keys and Sort Order***

Clicking on one of the **sort key** bevel buttons (Name or Date) causes the browser list to be sorted on that key. The sort order (ascending or descending) may be toggled by clicking the **sort order** bevel button. Navigation Services tracks the sort key and sort order for each application.

## ***Preview Area***

Navigation Services provides a **preview area** in all dialog boxes which open files. This area can be toggled on or off by the user using the Show/Hide Preview push button. If the preview area is visible, Navigation Services will automatically display a preview of any file that contains a valid 'pnot' resource. You can request preview display by ensuring that the `kNavAllowPreviews` constant is set in the `dialogOptionFlags` field of the `NavDialogOptions` structure (see below).

## ***Persistence***

**Persistence** is the ability of Navigation Services to store information, and to store it on a per-application basis. For example, when a primary dialog box is displayed, the browser defaults to the directory location that was in use when that particular dialog box was last closed by the application. In addition, if a file or folder was selected when the dialog box was last closed, that file or folder is automatically selected when the dialog is re-opened. The size, position, sort key and sort order of dialog boxes are also memorised for each application.

## Default Location and Selection

---

If you pass NULL in the `defaultLocation` parameter of functions such as `NavGetFile` (see below), Navigation Services will display the **default location and selection**. You can override the default location and selection of any Navigation Services dialog box by passing a pointer to an Apple event descriptor (AEDesc) for the new location in the `defaultLocation` parameter.

## Opening Files

---

The function `NavGetFile` displays an Open dialog box that prompts the user to select a file (or files) to open.

## Opening Multiple Files

---

By Shift-clicking in the Open dialog box's browser list, or by choosing the Select All item in the application's Edit menu, the user can open multiple files.<sup>2</sup>

## The Show Pop-up Menu

---

The Show pop-up menu in the Open dialog box allows the user to choose the **file types** to be displayed by the browser list and opened by Navigation Services. The list of available file types is built from information supplied by your application when it calls `NavGetFile`, and by services in the Translation Manager. (See Fig 3.)

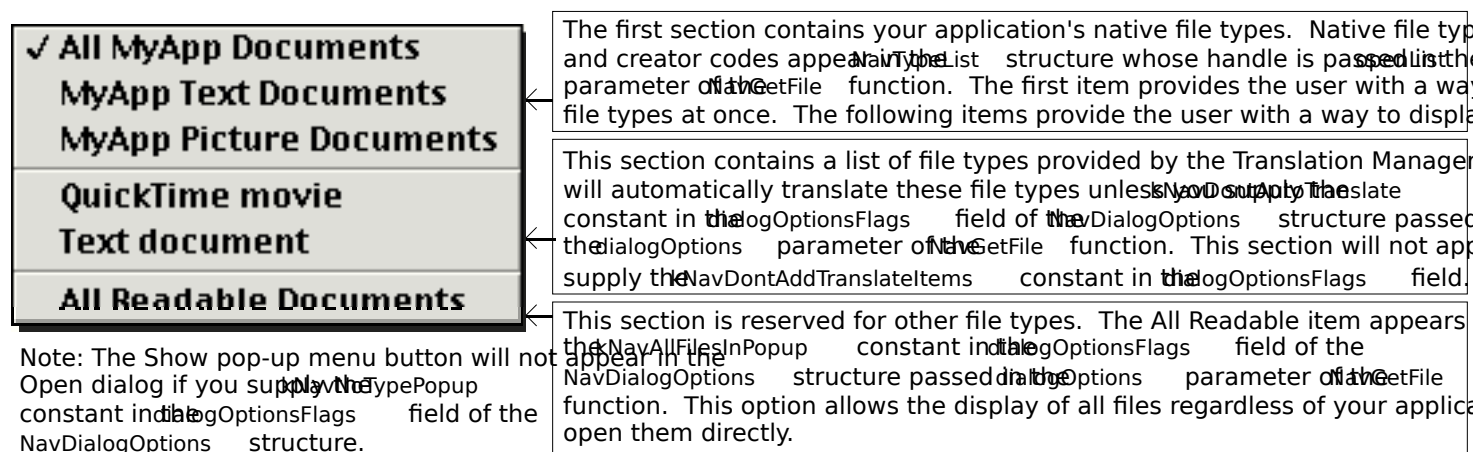


FIG 3 - THE SHOW POP-UP MENU AND FILE TYPE OPTIONS

## Native File Types Section

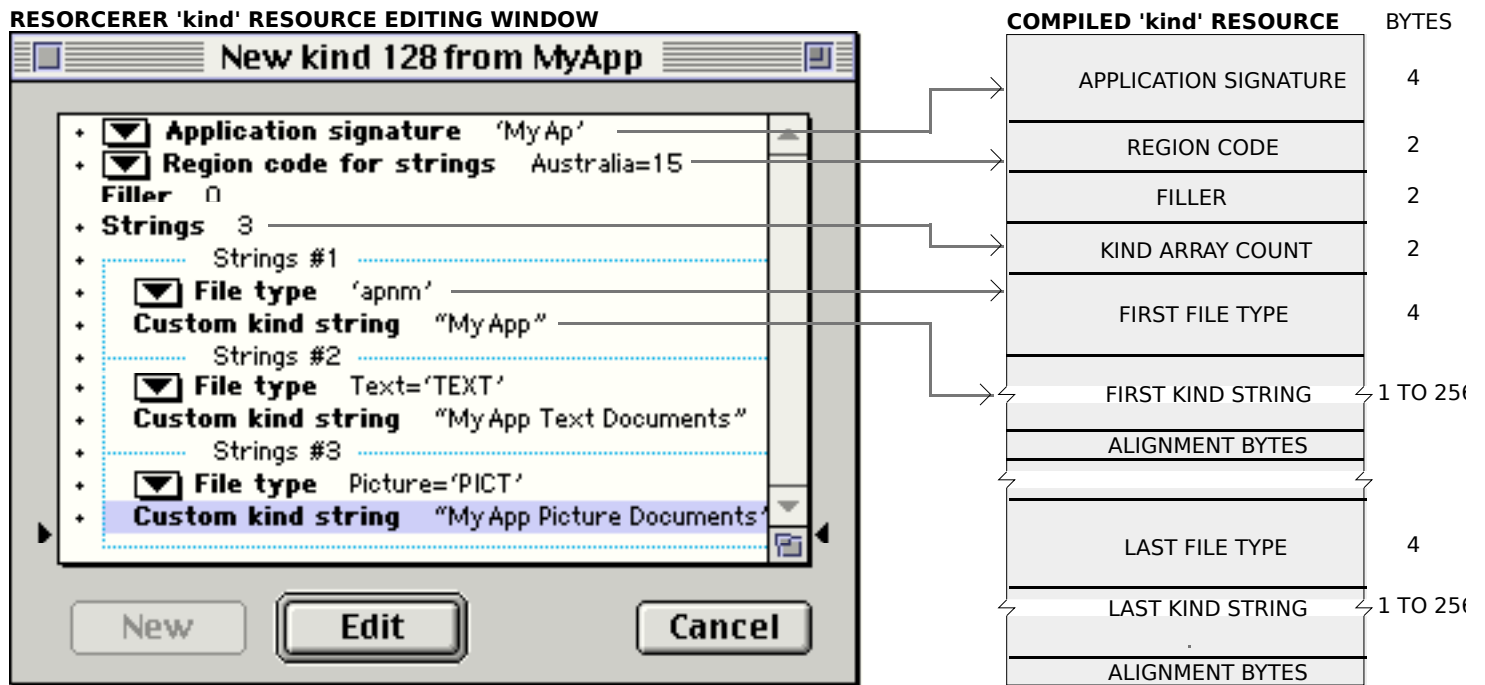
---

The first item in the **native file types** section of the Show pop-up menu defaults to All Readable Files if you do not assign the name of your application to the `clientName` field of the `NavDialogOptions` structure (see below) passed in the `dialogOptions` parameter of the `NavGetFile` function.

The remaining items in the native file types section will default to <Application Name> Document unless you provide **kind strings** to describe the file types included in your `NavTypeList` structure. You can do this by including a **kind resource** (a resource of type 'kind') in your application's resource fork. Fig 4 shows the structure of a compiled 'kind' resource and such a resource being created using Resorcerer.<sup>3</sup>

<sup>2</sup> Folders and volumes cannot be multiple-selected.

<sup>3</sup> The kind strings from your application's 'kind' resource also appear in the Kind column in Finder window list views.



Note: The special file type 'apnm' has been included so that, whenever Navigation Services encounters a document that type has not been included in the 'kind' resource, a kind string in the form "<application name> document" will be generated.

FIG 4 - STRUCTURE OF A COMPILED 'kind' RESOURCE AND CREATING A 'kind' RESOURCE

## The NavDialogOptions and NavTypeList Structures

Fig 3 referred to the NavTypeList structure, which defines a list of file types that your application is capable of opening, and the NavDialogOptions structure, which contains dialog box configuration settings.

### The NavTypeList Structure

The NavTypeList structure is as follows:

```

struct NavTypeList
{
    OSType    componentSignature;    // Your application signature.
    short     reserved;
    short     osTypeCount;          // How many file types will be defined.
    OSType    osType[1];           // A list of file types your application can open.
};

```

You can create your file type list dynamically or you can use an 'open' resource. Fig 5 shows the structure of a compiled 'open' resource and such a resource being created using Resorcerer.

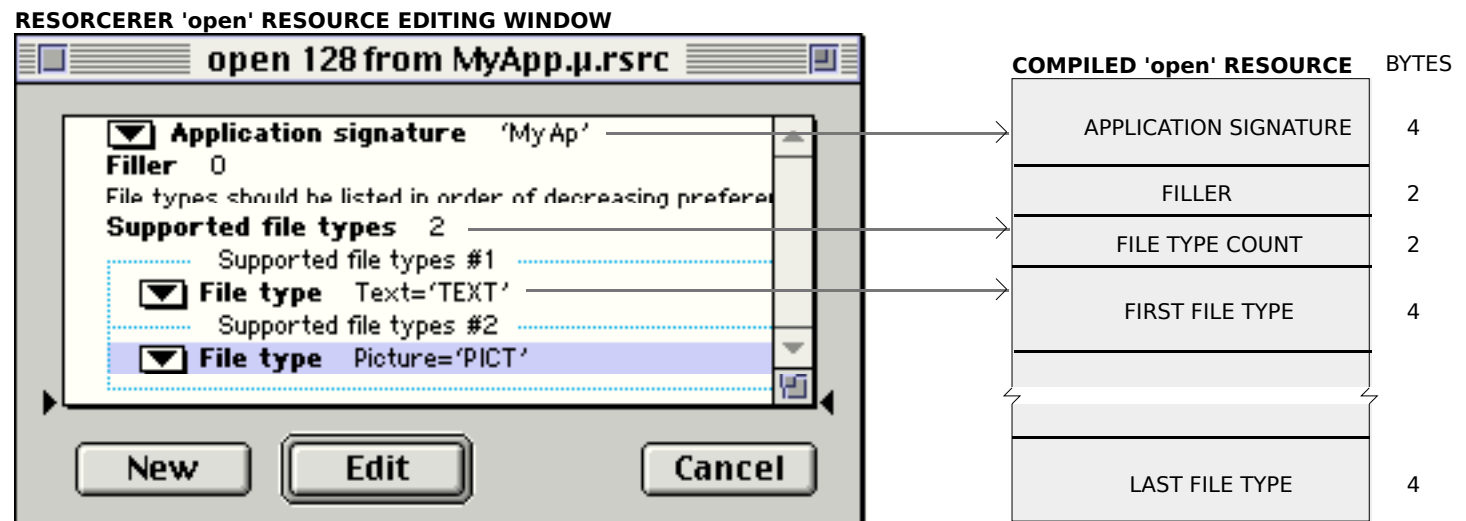


FIG 5 - STRUCTURE OF A COMPILED 'open' RESOURCE AND CREATING AN 'open' RESOURCE

## The NavDialogOptions Structure

The NavDialogOptions structure is as follows:

```
struct NavDialogOptions
{
    UInt16          version;
    NavDialogOptionFlags dialogOptionFlags;
    Point          location;
    Str255         clientName;
    Str255         windowTitle;
    Str255         actionButtonLabel;
    Str255         cancelButtonLabel;
    Str255         savedFileName;
    Str255         message;
    UInt32         preferenceKey;
    Handle         popupExtension;
};
```

### Field Descriptions

dialogOptionFlags One of the following constants of type NavDialogOptionFlags:

<b>Constant</b>	<b>Description</b>
kNavDefaultNavDlogOptions	Use default options. The defaults are as follows: <ul style="list-style-type: none"><li>• No custom control titles.</li><li>• No banner or prompt message.</li><li>• Automatic resolution of aliases.</li><li>• Support for file previews.</li><li>• No invisible file objects are displayed.</li><li>• Support for stationery.</li></ul>
kNavNoTypePopup	Don't show file type pop-up.
kNavDontAutoTranslate	Don't auto-translate on Open.
kNavDontAddTranslateltems	Don't add translation choices.
kNavAllFilesInPopup	Add "All Files" menu item.
kNavAllowStationery	Allow stationery files.
kNavAllowPreviews	Allow previews.
kNavAllowMultipleFiles	Allow multiple selection.
kNavAllowInvisibleFiles	Show invisible objects.
kNavDontResolveAliases	Don't resolve aliases.
kNavSelectDefaultLocation	Make default location the browser selection.
kNavSelectAllReadableItem	Make All Readable Items default selection.
kNavSupportPackages	Recognise file system packages.
kNavAllowOpenPackages	Allow opening of packages.
kNavDontAddRecents	Don't add chosen objects to Recents list.
kNavDontUseCustomFrame	Don't add the bevelled custom frame.

Those appearing on a gray background are available only in Navigation Services Version 2.0 or later. Version 2.0 was first issued with Mac OS 9.0.

location The upper-left location of the dialog box, in global coordinates. If the dialogOptionFlags field is NULL or the coordinate value is (-1,-1), then the dialog box appears in the same location as when last closed. The size and location of the dialog box is persistent, but defaults to opening in the middle of the main screen if any portion is not visible when opened at the persistent location and size.

clientName A string that identifies your application in the dialog box window title.

windowTitle	A string that you can provide to override the default window title.
actionButtonLabel	An alternative button title for the dialog box's action push button. If you do not specify a label, the push button will use the default label (Open or Save, for example.)
cancelButtonLabel	An alternative button title for the Cancel push button in dialog boxes.
savedFileName	The default file name for a saved file.
message	The string for the banner, or prompt, below the browser list. This message can provide more descriptive instructions for the user. If you don't provide a message string, the browser list will expand to fill that area.
preferenceKey	An application-defined value that identifies which set of dialog box preferences Navigation Services should use. If your application maintains multiple sets of preferences for a particular type of dialog box, you can determine which set is active by specifying the appropriate value in the <code>preferenceKey</code> field. For example, an application may allow one set of preferences when it calls the function <code>NavGetFile</code> to open text files and a different set of preferences when opening movie files. If you do not wish to provide a preference key, specify zero for the <code>preferenceKey</code> value.
popupExtension	A handle to one or more structures of type <code>NavMenuItemSpec</code> used to add extra menu items to the Show pop-up menu in an Open dialog box or the Format pop-up menu in Save dialog boxes. Using <code>NavMenuItemSpec</code> structures allows your application to add additional document types to be opened or saved, or different ways of saving a file (with or without line breaks, for example).

The function `NavGetDefaultDialogOptions` may be called to initialise a structure of type `NavDialogOptions` with the default dialog box options.

## ***The NavReplyRecord Structure***

---

The second parameter in the `NavGetFile` function, and in other Navigation Services functions, is a pointer to a structure of type `NavReplyRecord`. Navigation Services uses this structure to provide your application with information about the user's actions. The `NavReplyRecord` structure is as follows:

```

struct NavReplyRecord
{
    UInt16          version;
    Boolean          validRecord;
    Boolean          replacing;
    Boolean          isStationery;
    Boolean          translationNeeded;
    AEDescList      selection;
    ScriptCode      keyScript;
    FileTranslationSpec **fileTranslation;
};

```

### ***Field Descriptions***

validRecord	true if the user closes a dialog box by pressing Return or Enter, or by pressing the default button in an Open or Save dialog box. If this field is false, all other fields are unused and do not contain valid data.
replacing	true if the user chooses to save a file by replacing an existing file (thereby necessitating the removal or renaming of the existing file).

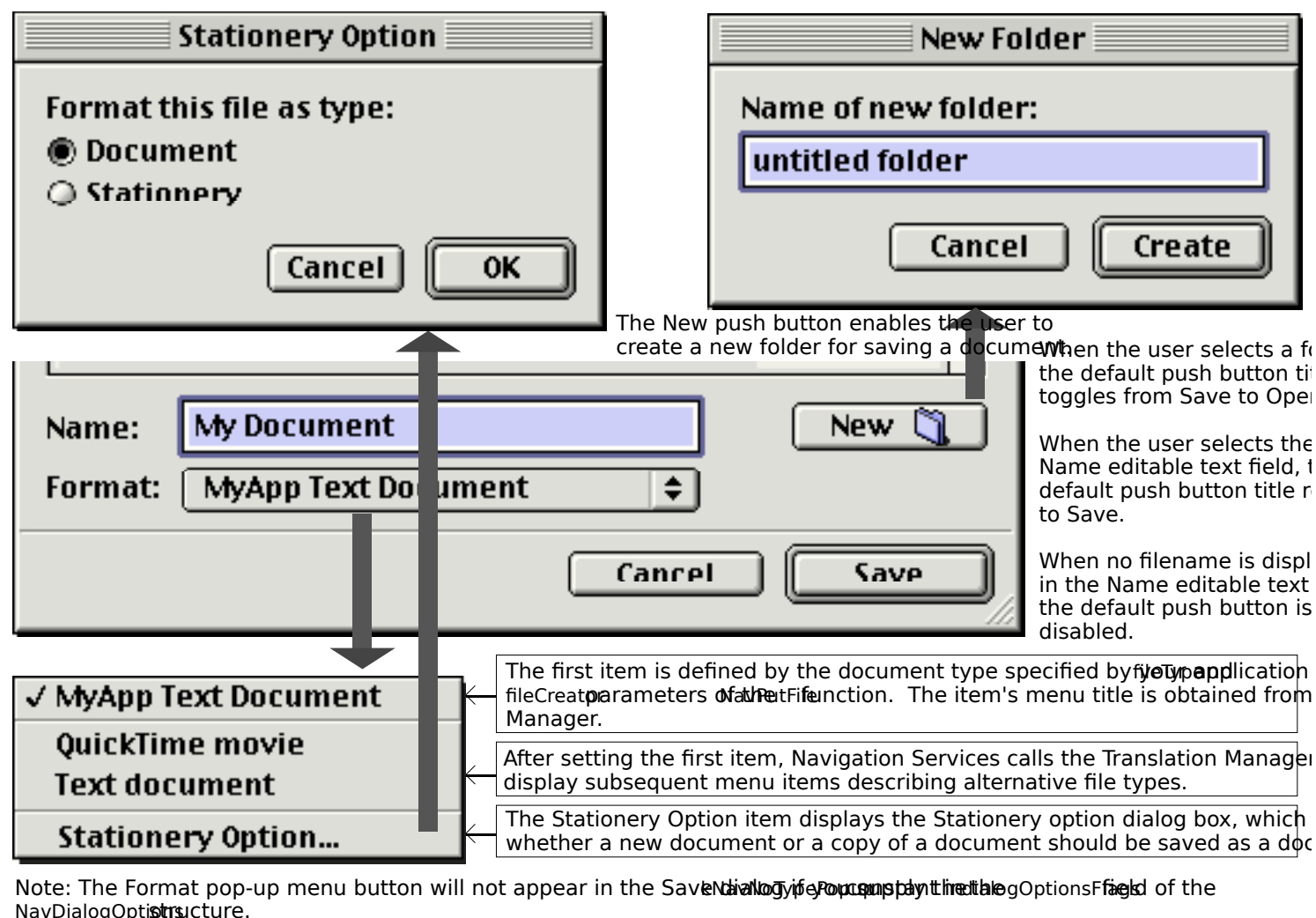
isStationery	A Boolean value informing your application that the file about to be saved should be saved as a stationery document.
translationNeeded	A Boolean value indicating that translation was or will be needed for files selected in Open and Save dialog boxes.
0	An Apple event descriptor list (AEDescList) created from FSSpec references to items selected through the dialog box. Navigation Services creates this list, which is automatically disposed of when your application calls the function NavDisposeReply (see below). You can determine the number of items in the list by calling the Apple Event Manager function AECountItems. (Some dialog boxes may return one or more items; a Save dialog box will always return one.) Each selected item is described in an AEDesc structure by the descriptor type typeFSS. You can coerce this descriptor into an FSSpec structure to perform operations such as opening the file.
keyScript	The keyboard script system used for the filename.
fileTranslation	A handle to a FileTranslationSpec structure. This structure contains a corresponding translation array for each file reference returned in the selection field. When opening files, Navigation Services will perform the translation automatically unless you set the kNavDontAutoTranslate flag in the dialogOptionFlags field of the NavDialogOptions structure. When Navigation Services performs an automatic translation, the FileTranslationSpec structure is strictly for the Translation Manager's use. If you turn off automatic translation, your application may use the FileTranslationSpec structure for its own translation scheme. If the user chooses a translation for a saved file, the FileTranslationSpec structure contains a single translation reference for the saved file and the translationNeeded field of the NavReplyRecord structure is set to true. The handle to the FileTranslationSpec structure is locked, so you can safely use dereferenced pointers.

When your application has finished using this structure, it should dispose of it by calling the function NavDisposeReply.



## Saving Files

The function `NavPutFile` displays the Save dialog box (see Fig 6).



**FIG 6 - THE SAVE DIALOG BOX (PARTIAL), THE FORMAT POP-UP MENU, THE STATIONERY OPTION DIALOG BOX, AND THE NEW FOLDER DIALOG BOX**

You should always call the function `NavCompleteSave` to complete any save operation. Amongst other things, `NavCompleteSave` performs any needed translation.

## Translating Files on Save

As stated at Fig 6, your application supplies its default file type and creator for saved files to the function `NavPutFile` and Navigation Services uses this information to build a pop-up menu of available translation choices obtained from the Translation Manager.

If the user selects an output file type that is different from the native type, Navigation Services prepares a translation specification and supplies a handle to it in the `fileTranslation` field of a `NavReplyRecord` structure. If you choose to provide your own translation, Navigation Services informs you that translation is required by setting the `translationNeeded` field of the `NavReplyRecord` structure to `true`.

If you wish to turn off automatic translation, set the value of the `translationNeeded` field of the `NavReplyRecord` structure to `false` before calling the `NavCompleteSave` function.

By default, the `NavPutFile` function saves translations as a copy of the original file. Your application can direct Navigation Services to replace the original with the translation by passing the `kNavTranslateInPlace` constant in the `howToTranslate` parameter of the `NavCompleteSave` function.

## Choosing A Folder

---

The function `NavChooseFolder` displays a dialog box that prompts the user to choose a folder (see Fig 7.)

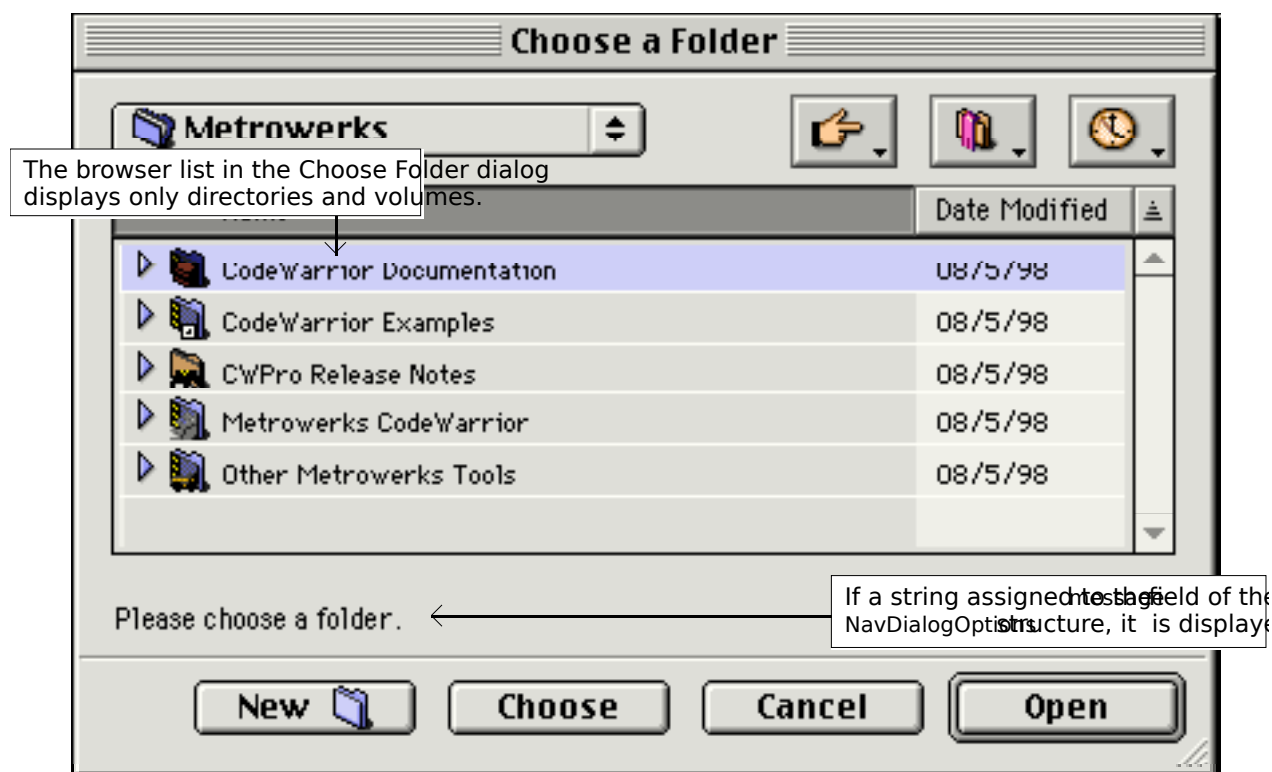


FIG 7 - CHOOSE A FOLDER DIALOG BOX

Several Navigation Services functions return Apple event descriptors for file objects. When Navigation Services passes your application an `AEDesc` structure of type `typeFSS` describing a file, the `name` field will contain the file's name and the `parID` field will contain the directory ID of the file's parent directory. However, when Navigation Services passes your application an `AEDesc` structure of type `typeFSS` describing a directory, the `name` field is empty and the `parID` field contains the directory ID of that directory.<sup>4</sup>

## Choosing Volumes, Files, File Objects, and Creating a New Folder

---

The function `NavChooseVolume` displays a dialog box that prompts the user to choose a volume.

The function `NavChooseFile` displays a dialog box that prompts the user to choose a file. This file can be a preferences file, dictionary, or other specialised file.

The function `NavChooseObject` displays a dialog box that prompts the user to choose a file object. This function is useful when you need the user to select an object which might be one of several different types.

The function `NavNewFolder` displays a dialog box that prompts the user to create a new folder.

As with the Choose Folder dialog box, these dialog boxes will display a string assigned to the `message` field of the `NavDialogOptions` structure immediately below the browser list.

<sup>4</sup> This means, incidentally, that you can use the `name` field to determine whether an object is a file or a folder.

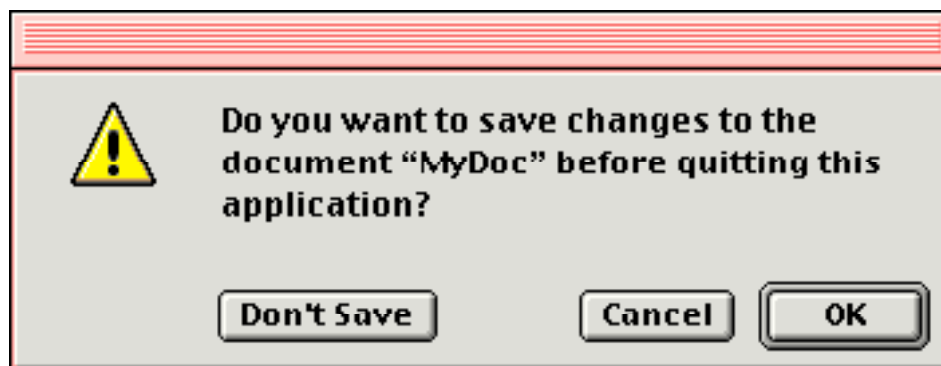
## ***Save Changes and Discard Changes Alert Boxes***

---

### ***Save Changes Alert Box***

---

To display a standard Save Changes alert box, your application passes the document title to the function `NavAskSaveChanges`, which creates an alert box similar to that shown at Fig 8.



**FIG 8 - STANDARD SAVE CHANGES ALERT BOX (QUITTING)**

One of the following constants is passed in the `action` parameter of the `NavAskSaveChanges` function:

```
kNavSaveChangesClosingDocument = 1
kNavSaveChangesQuittingApplication = 2
kNavSaveChangesOther = 0
```

After the user closes the alert box, Navigation Services tells your application which push button the user clicked by returning one of the following constants:

```
kNavAskSaveChangesSave = 1
kNavAskSaveChangesCancel = 2
kNavAskSaveChangesDontSave = 3
```

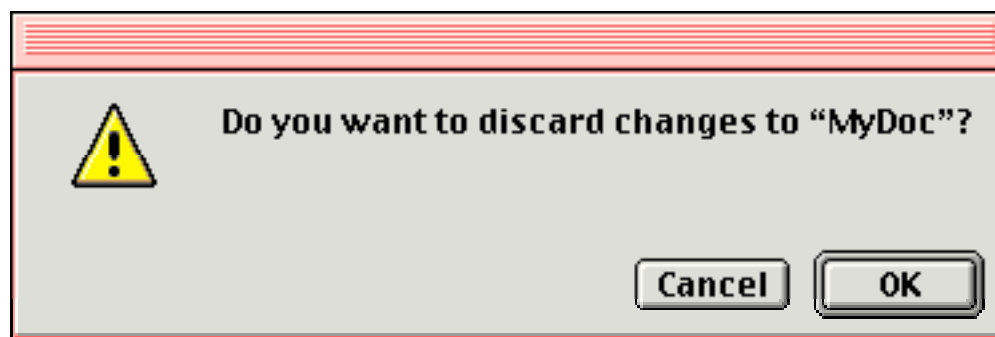
You can display a customised Save Changes alert box using the function `NavCustomAskSaveChanges`. A typical customised Save Changes alert box might contain text such as "You have not saved your work for ten minutes. Do you want to save now?". The `message` field of the `NavDialogOptions` structure passed in the `dialogOptions` parameter of `NavCustomAskSaveChanges` is the only one you must supply with a value.

Save Changes alert boxes are movable only if your application provides an application-defined event-handling callback function to handle update events.

### ***Discard Changes Alert Box***

---

To support a Revert To Saved item in your application's File menu, Navigation Services provides the Discard Changes alert box (see Fig 9), which is created by the function `NavAskDiscardChanges`.



**FIG 9 - DISCARD CHANGES ALERT BOX**

After the user closes the alert box, Navigation Services tells your application which button the user clicked by returning one of the following constants:

```
kNavAskDiscardChanges = 1  
kNavAskDiscardChangesCancel = 2
```

Discard Changes alert boxes are movable only if your application provides an application-defined event-handling callback function to handle update events.

## ***Application-Defined Functions***

---

### ***Application-Defined Event Handling***

---

As previously stated, Navigation Services primary dialog boxes are movable and resizable, and Navigation Services alert boxes are movable, only if your application provides an application-defined event-handling function to handle update events. A universal procedure pointer to your event-handling function is passed in the `eventProc` parameter of Navigation Services functions such as `NavGetFile` and `NavAskSaveChanges`.

The following is an example of an event-handling function:

```
pascal void myEventFunction(NavEventCallbackMessage callBackSelector,  
                           NavCBRecPtr callBackParms,void *callBackUD)  
{  
    WindowPtr window;  
  
    WindowPtr = (WindowPtr) callBackParms->eventData.event->message;  
  
    switch (callBackSelector)  
    {  
        case kNavCBEvt:  
            switch (callBackParms->eventData.event->what)  
            {  
                case updateEvt:  
                    doUpdate(window,(EventRecord *) callBackParms->eventData.event);  
                    break;  
            }  
            break;  
    }  
}
```

The formal parameter `callBackSelector` receives an **event message constant** indicating which type of call Navigation Services is making to the event-handling function. One such constant is `kNavCBEvt`, which indicates that an event has occurred, and which is the only message that needs to be processed by applications that do not customise the Open and Save dialog boxes. `callBackParms` is a pointer to a structure of type `NavCBRec`. The event's event structure resides in the `eventData` field of the `NavCBRec` structure.

### ***Application-Defined Object Filtering***

---

The process of choosing which files, folders and volumes to display in the browser list and the ShortCuts, Favourites, and Recent menus is known as object filtering. If your application needs simple, straightforward object filtering, and as previously described, you simply pass a pointer to a structure of type `NavTypeList` to the relevant Navigation Services function. If you desire more specific filtering, Navigation Services lets you implement an application-defined filter function. Filter functions give you more control over what can and cannot be displayed.

You can use both an `NavTypeList` structure and a filter function if you wish, but be aware that your filter function is directly affected by the `NavTypeList` structure. For example, if the `NavTypeList` structure contains only TEXT and PICT types, only TEXT and PICT files will be passed into your filter function.

You pass a universal procedure pointer to your filter function in the `filterProc` parameter of functions such as `NavGetFile`. Your filter function should return `true` if an object is to be displayed.

The following is an example of a simple filter function:

```
pascal void myFilterFunction(AEDesc* theItem,void* info,
                           NavCallBackUserData callBackUD,
                           NavFilterModes filterMode )
{
    OSErr theErr      = noErr;
    Boolean display   = true;
    NavFileOrFolderInfo* theInfo = (NavFileOrFolderInfo*) info;

    if(theItem->descriptorType == typeFSS)
        if(!theInfo->isFolder)
            if(theInfo->fileAndFolder.fileInfo.finderInfo.fdType != 'TEXT')
                display = false;

    return display;
}
```

## ***Application-Defined Previews***

---

To override how previews are drawn and handled, you can create a preview function and pass a universal procedure pointer to it in the `previewProc` parameter of Navigation Services functions such as `NavGetFile`. Your preview function must be defined like this:

```
pascal Boolean myPreviewProc(NavCBRecPtr callBackParms,NavCallBackUserData callBackUD);
```

`callBackParms`     A pointer to a `NavCBRec` structure that contains event data needed for your function to draw the preview.

`callBackUD`        A value set by your application. When the `NavGetFile` function calls back your event-handling function, the `callBackUD` value is passed back to your application.

**Return:** `true` if your preview function successfully draws the file preview. If your preview function returns `false`, Navigation Services will display the preview if the file contains a valid 'pnot' resource.

Your application can use the function `NavCustomControl` to determine if the preview area is visible and, if so, what its dimensions are.

## ***Adding Controls to a Navigation Services Dialog Box***

---

To add controls to a Navigation Services dialog box, you should:

- Provide an event-handling function to communicate with Navigation Services.
- Within the event-handling function, respond to the `kNavCBCustomize` event message constant, which your application can obtain from the `param` field of the `NavCBRec` structure. (See Application-Defined Event Handling, above.) The `customRect` field of the `NavCBRec` structure defines a rectangle in the local coordinates of the window. The top-left coordinates of this rectangle define the anchor point for the **customisation rectangle**, which is the area Navigation Services provides for your application to add custom dialog items. Your application responds by passing the values which will complete the dimensions of your required customisation rectangle. Navigation Services inspects the `customRect` field to determine if the requested dimensions can be accommodated in the screen space available. If not, the rectangle will be set to the largest size that can be accommodated and your application will be sent another `kNavCBCustomize` constant. Your application can continue to negotiate by examining the `customRect` field and requesting a different size

until Navigation Services provides an acceptable rectangle value. The minimum dimensions for the customisation area are 400 pixels wide by 40 pixels high.<sup>5</sup>

- With the customisation rectangle established, check for the `kNavCBStart` event message constant in the `param` field of the `NavCBRec` structure. `kNavCBStart` indicates that Navigation Services is opening the dialog box. After you obtain this constant, you can add your interface elements to the customisation rectangle. You can do this by providing a 'DITL' resource (in local coordinates, relative to the anchor point of the customisation rectangle) and passing the `kNavCtlAddControlList` constant in the `selector` parameter of the function `NavCustomControl`. The following shows one way to do this:

```
gDitlList = GetResource('DITL',kControlListID);
theErr = NavCustomControl(callBackParms->context,kNavCtlAddControlList,gDitlList);
```

The advantage of using a 'DITL' resource is that the Dialog Manager will handle all control movement and tracking.

- When Navigation Services supplies the `kNavCBTerminate` event message constant in the `param` field of the `NavCBRec` structure (after the user closes the dialog box), dispose of the control or resource.

## Main Navigation Services Constants, Data Types, and Functions

---

In the following, those constants and functions appearing on a gray background are available only in Navigation Services Version 2.0 or later. Version 2.0 was first issued with Mac OS 9.0.

### Constants

---

#### Configuration Options

<code>kNavDefaultNavDlogOptions</code>	<code>= 0x000000E4</code>	// Use defaults for all the options
<code>kNavNoTypePopup</code>	<code>= 0x00000001</code>	// Don't show file type/extension popup on Open/Save
<code>kNavDontAutoTranslate</code>	<code>= 0x00000002</code>	// Don't automatically translate on Open
<code>kNavDontAddTranslateItems</code>	<code>= 0x00000004</code>	// Don't add translation choices on Open/Save
<code>kNavAllFilesInPopup</code>	<code>= 0x00000010</code>	// "All Files" menu item in the type popup on Open
<code>kNavAllowStationery</code>	<code>= 0x00000020</code>	// Allow saving of stationery files
<code>kNavAllowPreviews</code>	<code>= 0x00000040</code>	// Allow to show previews
<code>kNavAllowMultipleFiles</code>	<code>= 0x00000080</code>	// Allow multiple items to be selected
<code>kNavAllowInvisibleFiles</code>	<code>= 0x00000100</code>	// Allow invisible items to be shown
<code>kNavDontResolveAliases</code>	<code>= 0x00000200</code>	// Don't resolve aliases
<code>kNavSelectDefaultLocation</code>	<code>= 0x00000400</code>	// Make the default location the browser selection
<code>kNavSelectAllReadableItem</code>	<code>= 0x00000800</code>	// Make dialog select "All Readable Documents" on Open
<code>kNavSupportPackages</code>	<code>= 0x00001000</code>	// Recognize file system packages
<code>kNavAllowOpenPackages</code>	<code>= 0x00002000</code>	// Allow opening of packages
<code>kNavDontAddRecents</code>	<code>= 0x00004000</code>	// Don't add chosen objects to the recents list
<code>kNavDontUseCustomFrame</code>	<code>= 0x00008000</code>	// Don't add the bevelled custom frame

#### Save Changes Request

<code>kNavSaveChangesClosingDocument</code>	<code>= 1</code>
<code>kNavSaveChangesQuittingApplication</code>	<code>= 2</code>
<code>kNavSaveChangesOther</code>	<code>= 0</code>

#### Save Changes Action

<code>kNavAskSaveChangesSave</code>	<code>= 1</code>
<code>kNavAskSaveChangesCancel</code>	<code>= 2</code>
<code>kNavAskSaveChangesDontSave</code>	<code>= 3</code>

#### Discard Changes Action

<sup>5</sup> The `customRect` field contains an empty rectangle if customisation is not allowed (i.e. dialog boxes other than Open or Save). In this case, your application can simply ignore the call.

```
kNavAskDiscardChanges      = 1
kNavAskDiscardChangesCancel = 2
```

## Event Message

```
kNavCBEvent                = 0           // An event has occurred (update, idle events, etc.)
kNavCBCustomize            = 1           // Protocol for negotiating customization space
kNavCBStart                = 2           // The navigation dialog is starting up
kNavCBTerminate           = 3           // The navigation dialog is closing down
kNavCBAdjustRect          = 4           // The navigation dialog is being resized
kNavCBNewLocation         = 5           // User has chosen a new location in the browser
kNavCBShowDesktop         = 6           // User has navigated to the desktop
kNavCBSelectEntry         = 7           // User has made a selection in the browser
kNavCBPopupMenuSelect     = 8           // Signifies that a popup menu selection was made
kNavCBAccept              = 9           // User has accepted the navigation dialog
kNavCBCancel              = 10          // User has cancelled the navigation dialog
kNavCBAdjustPreview       = 11         // Preview button clicked or preview was resized
kNavCBOpenSelection       = (long) 0x80000000 // User has opened or chosen an object
```

## Object Filtering

```
kNavFilteringBrowserList  = 0
kNavFilteringFavorites    = 1
kNavFilteringRecents      = 2
kNavFilteringShortCutVolumes = 3
```

## Data Types

---

### NavDialog Options Structure

```
struct NavDialogOptions
{
    UInt16          version;
    NavDialogOptionFlags dialogOptionFlags;
    Point           location;
    Str255          clientName;
    Str255          windowTitle;
    Str255          actionButtonLabel;
    Str255          cancelButtonLabel;
    Str255          savedFileName;
    Str255          message;
    UInt32          preferenceKey;
    Handle          popupExtension;
    char            reserved[494];
};
typedef struct NavDialogOptions NavDialogOptions;
```

### Nav Reply Structure

```
struct NavReplyRecord
{
    UInt16          version;
    Boolean          validRecord;
    Boolean          replacing;
    Boolean          isStationery;
    Boolean          translationNeeded;
    AEDescList      selection;
    ScriptCode      keyScript;
    FileTranslationSpec **fileTranslation;
    UInt32          reserved1;
    char            reserved[231];
};
typedef struct NavReplyRecord NavReplyRecord;
```

### NavTypeList

```
struct NavTypeList
{
    OSType          componentSignature;
    short           reserved;
    short           osTypeCount;
    OSType          osType[1];
};
typedef struct NavTypeList NavTypeList;
typedef NavTypeList *NavTypeListPtr;
```

```
typedef NavTypeListPtr *NavTypeListHandle;
```

## Functions

---

### Calling Navigation Services

```
OSErr NavLoad(void);
OSErr NavUnload(void);
OSErr NavGetDefaultDialogOptions(NavDialogOptions *dialogOptions);
OSErr NavDisposeReply(NavReplyRecord *reply);
```

### Choosing Files, Folders, and Volumes

```
OSErr NavGetFile(AEDesc *defaultLocation,NavReplyRecord *reply,
NavDialogOptions *dialogOptions,NavEventUPP eventProc,NavPreviewUPP previewProc,
NavObjectFilterUPP filterProc,NavTypeListHandle typeList,
NavCallBackUserData callBackUD);
OSErr NavChooseFile(AEDesc *defaultLocation,NavReplyRecord *reply,
NavDialogOptions *dialogOptions,NavEventUPP eventProc,NavPreviewUPP previewProc,
NavObjectFilterUPP filterProc,NavTypeListHandle typeList,
NavCallBackUserData callBackUD);
OSErr NavChooseVolume(AEDesc *defaultSelection,NavReplyRecord *reply,
NavDialogOptions *dialogOptions,NavEventUPP eventProc,NavObjectFilterUPP filterProc,
NavCallBackUserData callBackUD);
OSErr NavChooseFolder(AEDesc *defaultLocation,NavReplyRecord *reply,
NavDialogOptions *dialogOptions,NavEventUPP eventProc,NavObjectFilterUPP filterProc,
NavCallBackUserData callBackUD);
OSErr NavChooseObject(AEDesc *defaultLocation,NavReplyRecord *reply,
NavDialogOptions *dialogOptions,NavEventUPP eventProc,NavObjectFilterUPP filterProc,
NavCallBackUserData callBackUD);
OSErr NavNewFolder(AEDesc *defaultLocation,NavReplyRecord *reply,
NavDialogOptions *dialogOptions,NavEventUPP eventProc,NavCallBackUserData callBackUD);
OSErr NavCreatePreview(AEDesc *theObject,OSType previewDataType,const void *previewData,
Size previewDataSize);
```

### Saving Files

```
OSErr NavPutFile(AEDesc *defaultLocation,NavReplyRecord *reply,
NavDialogOptions *dialogOptions,NavEventUPP eventProc,OSType fileType,
OSType fileCreator, NavCallBackUserData callBackUD);
OSErr NavAskSaveChanges(NavDialogOptions *dialogOptions,NavAskSaveChangesAction action,
NavAskSaveChangesResult *reply,NavEventUPP eventProc,NavCallBackUserData callBackUD);
OSErr NavCustomAskSaveChanges(NavDialogOptions *dialogOptions,
NavAskSaveChangesResult *reply,NavEventUPP eventProc,NavCallBackUserData callBackUD);
OSErr NavAskDiscardChanges(NavDialogOptions *dialogOptions,
NavAskDiscardChangesResult *reply,NavEventUPP eventProc,
NavCallBackUserData callBackUD);
OSErr NavCompleteSave(NavReplyRecord *reply,NavTranslationOptions howToTranslate);
```

## Demonstration Program

---

```
// ~~~~~
// Files2.h
// ~~~~~
//
// This program demonstrates:
//
// • The use of Navigation Services rather than the Standard File Package to display
//   Open, Save, and Choose a Folder dialog boxes.
//
// • The use of Navigation Services rather than the Dialog Manager to display Save
//   Changes and Discard Changes alert boxes.
//
// • Application-defined file handling functions associated with:
//
//   • The user invoking the File menu Open..., Close, Save, Save As..., Revert to Saved,
//     and Quit commands of a typical application.
//
//   • Handling of the required Apple events Open Application, Re-open Application,
//     Open Documents and Quit Application.
//
// These functions are essentially the same as those in the demonstration program
// Files1 except that the safe-save methodology used in Files1 is not used, all
```





```

    Boolean      windowTouched;
} docStructure, *docStructurePointer, **docStructureHandle;

```

```
typedef StandardFileReply *standardFileReplyPtr;
```

```
//
```

```
..... defines
```

```

#define mApple          128
#define iAbout          1
#define mFile           129
#define iNew            1
#define iOpen           2
#define iClose          4
#define iSave           5
#define iSaveAs         6
#define iRevert         7
#define iQuit           12
#define mDemonstration 131
#define iTouchWindow   1
#define iChooseAFolderDialog 3
#define rNewWindow      128
#define rMenuBar        128
#define rRevertAlert    128
#define rCloseFileAlert 129
#define rCustomOpenDialog 130
#define iPopuItem       10
#define rSelectDirectoryDialog 131
#define iSelectButton   10
#define rErrorStrings    128
#define eInstallHandler 1000
#define eMaxWindows      1001
#define eFilesOpen       opWrErr
#define eNoNavServices   1002
#define eCantFindFinderProcess 1003
#define rMiscStrings     129
#define sApplicationName 1
#define sChooseAFolder  2
#define rOpenResource    128
#define kMaxWindows      10
#define MAXLONG          0x7FFFFFFF
#define MIN(a,b)         ((a) < (b) ? (a) : (b))

```

```
/////
```

```
//
```

```
..... function prototypes
```

```

void      main                (void);
void      eventLoop           (void);
void      doInitManagers      (void);
void      doInstallAEHandlers (void);
void      doEvents            (EventRecord *);
void      doMouseDown         (EventRecord *);
void      doBringFinderToFront (void);
OSErr     doFindProcess        (OSType,OSType,ProcessSerialNumber *);
void      doUpdate            (EventRecord *);
void      doMenuChoice        (SInt32);
void      doFileMenu          (SInt16);
void      doAdjustMenus       (void);
void      doErrorAlert        (SInt16);
void      doCopyPString       (Str255,Str255);
void      doConcatPStrings    (Str255,Str255);
void      doTouchWindow       (void);
void      doSynchroniseFiles  (void);
pascal OSErr doOpenAppEvent    (AppleEvent *,AppleEvent *,SInt32);
pascal OSErr doReopenAppEvent  (AppleEvent *,AppleEvent *,SInt32);
pascal OSErr doOpenDocsEvent   (AppleEvent *,AppleEvent *,SInt32);
pascal OSErr doQuitAppEvent    (AppleEvent *,AppleEvent *,SInt32);
OSErr     doHasGotRequiredParams (AppleEvent *);

OSErr     doNewCommand          (void);
OSErr     doOpenCommand         (void);
OSErr     doCloseCommand        (NavAskSaveChangesAction);
OSErr     doSaveCommand         (void);
OSErr     doSaveAsCommand       (void);
OSErr     doRevertCommand       (void);
OSErr     doQuitCommand         (NavAskSaveChangesAction);
OSErr     doNewDocWindow        (Boolean,OSType);

```

```
/////
```

```
/////
```

```

OSErr doOpenFile          (FSSpec,OSType);
OSErr doReadTextFile      (WindowPtr);
OSErr doReadPictFile      (WindowPtr);
OSErr doCloseFile         (WindowPtr,docStructureHandle,NavAskSaveChangesAction);
OSErr doWriteFile         (WindowPtr,Boolean);
OSErr doWriteTextData     (WindowPtr,SInt16);
OSErr doWritePictData     (WindowPtr,SInt16);
OSErr doCopyResources     (WindowPtr);
OSErr doCopyAResource     (ResType,SInt16,SInt16,SInt16);
pascal void navEventFunction (NavEventCallbackMessage,NavCBRecPtr,NavCallBackUserData);

OSErr doChooseAFolderDialog (FSSpec *);

// ~~~~~
// Files2.c
// ~~~~~

//
..... includes

#include "Files2.h"

//
..... global variables

Boolean          gMacOS85Present = false;          ////
Boolean          gDone;
Boolean          gInBackground;
AEEEventHandlerUPP doOpenAppEventUPP;
AEEEventHandlerUPP doReopenAppEventUPP;
AEEEventHandlerUPP doOpenDocsEventUPP;
AEEEventHandlerUPP doQuitAppEventUPP;
SInt16          gAppResFileRefNum;

extern SInt16    gCurrentNumberOfWindows;
extern Rect      gDestRect,gViewRect;

// ~~~~~ main

void main(void)
{
    OSErr      osError;          ////
    SInt32     response;         ////
    Handle     menubarHdl;
    MenuHandle menuHdl;

    //
    .....initialise managers

    doInitManagers();

    // ..... check whether Mac OS 8.5 or later is
    present

    osError = Gestalt(gestaltSystemVersion,&response);          ////

    if(osError == noErr && response >= 0x00000850)          ////
        gMacOS85Present = true;          ////

    // ..... check for Navigation Services, and pre-load (optional)

    if(NavServicesAvailable())
        NavLoad();
    else
        doErrorAlert(eNoNavServices);

    // ..... create routine descriptors for Apple event handlers

    doOpenAppEventUPP = NewAEEEventHandlerProc((ProcPtr) doOpenAppEvent);
    doReopenAppEventUPP= NewAEEEventHandlerProc((ProcPtr) doReopenAppEvent);
    doOpenDocsEventUPP = NewAEEEventHandlerProc((ProcPtr) doOpenDocsEvent);
    doQuitAppEventUPP  = NewAEEEventHandlerProc((ProcPtr) doQuitAppEvent);

    // ..... set application's resource fork as current resource file

    gAppResFileRefNum = CurResFile();
}

```

```

// ..... set
up menu bar and menus

menubarHdl = GetNewMBar(rMenubar);
if(menubarHdl == NULL)
    doErrorAlert(MemError());
SetMenuBar(menubarHdl);
DrawMenuBar();

menuHdl = GetMenuHandle(mApple);
if(menuHdl == NULL)
    doErrorAlert(MemError());
else
    AppendResMenu(menuHdl,'DRVR');

// ..... install required Apple event
handlers

doInstallAEHandlers();

//
..... enter event loop

eventLoop();
}

// ..... eventLoop
void eventLoop(void)
{
    EventRecord eventStructure;

    gDone = false;

    while(!gDone)
    {
        if(WaitNextEvent(everyEvent,&eventStructure,MAXLONG,NULL))
            doEvents(&eventStructure);
#ifdef TARGET_CPU_PPC
            else
                if(gMacOS85Present)
                    doSynchroniseFiles();
#endif
    }
}

// ..... doInitManagers
void doInitManagers(void)
{
    MaxApplZone();
    MoreMasters();

    InitGraf(&qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(NULL);

    InitCursor();
    FlushEvents(everyEvent,0);

    RegisterAppearanceClient();
}

// ..... doInstallAEHandlers
void doInstallAEHandlers(void)
{
    OSErr osError;

    osError = AEInstallEventHandler(kCoreEventClass,kAEOpenApplication,doOpenAppEventUPP,
                                  0L,false);
    if(osError != noErr) doErrorAlert(eInstallHandler);

    osError = AEInstallEventHandler(kCoreEventClass,kAEReopenApplication,

```

```

doReopenAppEventUPP,0L,false);
if(osError != noErr) doErrorAlert(eInstallHandler);

osError = AERInstallEventHandler(kCoreEventClass,kAEOpenDocuments,doOpenDocsEventUPP,
                                0L,false);
if(osError != noErr) doErrorAlert(eInstallHandler);

osError = AERInstallEventHandler(kCoreEventClass,kAEQuitApplication,doQuitAppEventUPP,
                                0L,false);
if(osError != noErr) doErrorAlert(eInstallHandler);
}

// doEvents doEvents

void doEvents(EventRecord *eventStrucPtr)
{
    SInt8 charCode;

    switch(eventStrucPtr->what)
    {
        case kHighLevelEvent:
            AEProcessAppleEvent(eventStrucPtr);
            break;

        case mouseDown:
            doMouseDown(eventStrucPtr);
            break;

        case keyDown:
        case autoKey:
            charCode = eventStrucPtr->message & charCodeMask;
            if((eventStrucPtr->modifiers & cmdKey) != 0)
            {
                doAdjustMenus();
                doMenuChoice(MenuEvent(eventStrucPtr));
            }
            break;

        case updateEvt:
            doUpdate(eventStrucPtr);
            break;

        case osEvt:
            switch((eventStrucPtr->message >> 24) & 0x000000FF)
            {
                case suspendResumeMessage:
                    gInBackground = (eventStrucPtr->message & resumeFlag) == 0;
                    break;
            }
            HiliteMenu(0);
            break;
    }
}

// doMouseDown doMouseDown

void doMouseDown(EventRecord *eventStrucPtr)
{
    WindowPtr windowPtr;
    SInt16 partCode;
    OSStatus osStatus;
    Boolean handled = false;
    SInt32 itemSelected;

    partCode = FindWindow(eventStrucPtr->where,&windowPtr);

    switch(partCode)
    {
        case inMenuBar:
            doAdjustMenus();
            doMenuChoice(MenuSelect(eventStrucPtr->where));
            break;

        case inContent:
            if(windowPtr != FrontWindow())
                SelectWindow(windowPtr);
            break;

        case inGoAway:

```

```

////
////
////

```

```

        if(TrackGoAway(windowPtr,eventStrucPtr->where) == true)
            doCloseCommand(kNavSaveChangesClosingDocument);
        break;
#endif TARGET_CPU_PPC
    case inProxyIcon:
        osStatus = TrackWindowProxyDrag(windowPtr,eventStrucPtr->where);
        if(osStatus == errUserWantsToDragWindow)
            handled = false;
        else if(osStatus == noErr)
            handled = true;
#endif

    case inDrag:
#ifdef TARGET_CPU_PPC
        if(gMacOS85Present)
        {
            if(!handled)
            {
                if(IsWindowPathSelectClick(windowPtr,eventStrucPtr))
                {
                    if(WindowPathSelect(windowPtr,NULL,&itemSelected) == noErr)
                    {
                        if(LoWord(itemSelected) > 1)
                            doBringFinderToFront();
                    }

                    handled = true;
                }
            }
            if(!handled)
                DragWindow(windowPtr,eventStrucPtr->where,&qd.screenBits.bounds);
        }
    else
        DragWindow(windowPtr,eventStrucPtr->where,&qd.screenBits.bounds);
    break;
}
}

#ifdef TARGET_CPU_PPC
// doBringFinderToFront
void doBringFinderToFront(void)
{
    ProcessSerialNumber finderProcess;

    if(doFindProcess('MACS','FNDR",&finderProcess) == noErr)
        SetFrontProcess(&finderProcess);
    else
        doErrorAlert(eCantFindFinderProcess);
}

// doFindProcess
OSErr doFindProcess(OSType creator,OSType type,ProcessSerialNumber *outProcSerNo)
{
    ProcessSerialNumber procSerialNo;
    ProcessInfoRec      procInfoStruc;
    OSErr                osError = 0;

    procSerialNo.highLongOfPSN = 0;
    procSerialNo.lowLongOfPSN = kNoProcess;

    procInfoStruc.processInfoLength = sizeof(ProcessInfoRec);
    procInfoStruc.processName       = NULL;
    procInfoStruc.processAppSpec    = NULL;
    procInfoStruc.processLocation   = NULL;

    while(true)
    {
        osError = GetNextProcess(&procSerialNo);
        if(osError != noErr)
            break;

        osError = GetProcessInformation(&procSerialNo,&procInfoStruc);
        if(osError != noErr)
            break;
        if((procInfoStruc.processSignature == creator) && (procInfoStruc.processType == type))

```









```

if(errorCode == eInstallHandler)
    GetIndString(errorString,rErrorStrings,1);
else if(errorCode == eMaxWindows)
    GetIndString(errorString,rErrorStrings,2);
else if(errorCode == eFilesOpen)
    GetIndString(errorString,rErrorStrings,3);
else if(errorCode == eNoNavServices)
    GetIndString(errorString,rErrorStrings,4);
else if(errorCode == eCantFindFinderProcess)
    GetIndString(errorString,rErrorStrings,5);
else
{
    GetIndString(errorString,rErrorStrings,6);
    NumToString((SInt32) errorCode,theString);
    doConcatPStrings(errorString,theString);
}

if(errorCode != memFullErr && errorCode != eNoNavServices)
    StandardAlert(kAlertCautionAlert,errorString,NULL,&paramRec,&itemHit);
else
{
    StandardAlert(kAlertStopAlert,errorString,NULL,&paramRec,&itemHit);
    ExitToShell();
}
}

// doCopyPString
void doCopyPString(Str255 sourceString,Str255 destinationString)
{
    SInt16 stringLength;

    stringLength = sourceString[0];
    BlockMove(sourceString + 1,destinationString + 1,stringLength);
    destinationString[0] = stringLength;
}

// doConcatPStrings
void doConcatPStrings(Str255 targetString,Str255 appendString)
{
    SInt16 appendLength;

    appendLength = MIN(appendString[0],255 - targetString[0]);

    if(appendLength > 0)
    {
        BlockMoveData(appendString+1,targetString+targetString[0]+1,(SInt32) appendLength);
        targetString[0] += appendLength;
    }
}

// doTouchWindow
void doTouchWindow(void)
{
    WindowPtr windowPtr;
    docStructureHandle docStrucHdl;

    windowPtr = FrontWindow();
    docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);

    SetPort(windowPtr);

    TextSize(48);
    MoveTo(30,170);
    DrawString("\pWINDOW TOUCHED");
    TextSize(12);

    (*docStrucHdl)->windowTouched = true;

#ifdef TARGET_CPU_PPC
    if(gMacOS85Present)
        SetWindowModified(windowPtr,true);
#endif
}

// doSynchroniseFiles

```

```

#if TARGET_CPU_PPC
void doSynchroniseFiles(void)
{
    UInt32          currentTicks;
    WindowPtr      windowPtr;
    SInt16         trashVRefNum;
    SInt32         trashDirID;
    static UInt32  nextSynchTicks = 0;
    docStructureHandle docStrucHdl;
    Boolean        aliasChanged;
    AliasHandle    aliasHdl;
    FSSpec        newFSSpec;
    OSErr         osError;

    currentTicks = TickCount();
    windowPtr    = FrontNonFloatingWindow();

    if(currentTicks > nextSynchTicks)
    {
        while(windowPtr != NULL)
        {
            docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);

            if(docStrucHdl != NULL)
            {
                if((*docStrucHdl)->aliasHdl == NULL)
                    break;

                aliasChanged = false;
                aliasHdl = (*docStrucHdl)->aliasHdl;
                ResolveAlias(NULL,aliasHdl,&newFSSpec,&aliasChanged);

                if(aliasChanged)
                {
                    (*docStrucHdl)->fileFSSpec = newFSSpec;
                    SetWTitle(windowPtr,newFSSpec.name);
                }

                osError = FindFolder(kOnSystemDisk,kTrashFolderType,kDontCreateFolder,
                                     &trashVRefNum,&trashDirID);

                if(osError == noErr)
                {
                    do
                    {
                        if(newFSSpec.parID == fsRtParID)
                            break;

                        if((newFSSpec.vRefNum == trashVRefNum) && (newFSSpec.parID == trashDirID))
                        {
                            FSClose((*docStrucHdl)->fileRefNum);
                            if((*docStrucHdl)->editStrucHdl)
                                TEDispose((*docStrucHdl)->editStrucHdl);
                            if((*docStrucHdl)->pictureHdl)
                                KillPicture((*docStrucHdl)->pictureHdl);
                            DisposeHandle((Handle) docStrucHdl);
                            DisposeWindow(windowPtr);
                            gCurrentNumberOfWindows --;
                            break;
                        }
                    } while(FSMakeFSSpec(newFSSpec.vRefNum,newFSSpec.parID,"\p",&newFSSpec) == noErr);
                }
            }

            windowPtr = GetNextWindow(windowPtr);
        }

        nextSynchTicks = currentTicks + 15;
    }
}
#endif

// doOpenAppEvent
pascal OSErr doOpenAppEvent(AppleEvent *appEvent,AppleEvent *reply,SInt32 handlerRefCon)
{
    OSErr osError;
}

```

```

    osError = doHasGotRequiredParams(appEvent);
    if(osError == noErr)
        osError = doNewCommand();

    return(osError);
}

// ~~~~~ doReopenAppEvent

pascal OSErr doReopenAppEvent(AppleEvent *appEvent,AppleEvent *reply,
                             SInt32 handlerRefCon)
{
    OSErr osError;

    osError = doHasGotRequiredParams(appEvent);
    if(osError == noErr)
        if(!FrontWindow())
            osError = doNewCommand();

    return(osError);
}

// ~~~~~ doOpenDocsEvent

pascal OSErr doOpenDocsEvent(AppleEvent *appEvent,AppleEvent *reply,SInt32 handlerRefcon)
{
    FSSpec     fileSpec;
    AEDescList docList;
    OSErr      osError, ignoreErr;
    SInt32     index, numberOfItems;
    Size       actualSize;
    AEKeyword  keyWord;
    DescType   returnedType;
    FlInfo     fileInfo;

    osError = AEGetParamDesc(appEvent,keyDirectObject,typeAEList,&docList);

    if(osError == noErr)
    {
        osError = doHasGotRequiredParams(appEvent);
        if(osError == noErr)
        {
            AECntItems(&docList,&numberOfItems);
            if(osError == noErr)
            {
                for(index=1;index<=numberOfItems;index++)
                {
                    osError = AEGetNthPtr(&docList,index,typeFSS,&keyWord,&returnedType,
                                           (Ptr) &fileSpec,sizeof(fileSpec),&actualSize);

                    if(osError == noErr)
                    {
                        osError = FSpGetFlInfo(&fileSpec,&fileInfo);
                        if(osError == noErr)
                        {
                            if(osError = doOpenFile(fileSpec,fileInfo.fdType))
                                doErrorAlert(osError);
                        }
                    }
                    else
                        doErrorAlert(osError);
                }
            }
        }
        else
            doErrorAlert(osError);

        ignoreErr = AEDisposeDesc(&docList);
    }
    else
        doErrorAlert(osError);

    return(osError);
}

// ~~~~~ doQuitAppEvent

pascal OSErr doQuitAppEvent(AppleEvent *appEvent,AppleEvent *reply,SInt32 handlerRefcon)
{
    OSErr osError;

```



```

OSErr doOpenCommand(void)
{
    OSErr            osError            = noErr;
    NavDialogOptions dialogOptions;
    NavTypeListHandle fileTypeListHdl = NULL;
    NavEventUPP      navEventFunctionUPP;
    NavReplyRecord   navReplyStruc;
    SInt32           count, index;
    AEKeyword        theKeyword;
    DescType         actualType;
    FSSpec           fileSpec;
    Size             actualSize;
    FInfo            fileInfo;
    OSType           documentType;

    osError = NavGetDefaultDialogOptions(&dialogOptions);

    if(osError == noErr)
    {
        GetIndString((unsigned char*) &dialogOptions.clientName, rMiscStrings, sApplicationName);
        fileTypeListHdl = (NavTypeListHandle) GetResource('open', rOpenResource);

        navEventFunctionUPP = NewNavEventUPP(navEventFunction);
        fileTypeListHdl = (NavTypeListHandle) GetResource('open', rOpenResource);

        osError = NavGetFile(NULL, &navReplyStruc, &dialogOptions, navEventFunctionUPP,
                               NULL, NULL, fileTypeListHdl, NULL);

        DisposeNavEventUPP(navEventFunctionUPP);

        if(osError == noErr && navReplyStruc.validRecord)
        {
            osError = AECntItems(&(navReplyStruc.selection), &count);
            if(osError == noErr)
            {
                for(index=1; index<=count; index++)
                {
                    osError = AEGGetNthPtr(&(navReplyStruc.selection), index, typeFSS, &theKeyword,
                                           &actualType, &fileSpec, sizeof(fileSpec), &actualSize);

                    if((osError = FSpGetFInfo(&fileSpec, &fileInfo)) == noErr)
                    {
                        documentType = fileInfo.fdType;
                        osError = doOpenFile(fileSpec, documentType);
                    }
                }
            }

            osError = NavDisposeReply(&navReplyStruc);
        }

        if(fileTypeListHdl != NULL)
            ReleaseResource((Handle) fileTypeListHdl);
    }

    if(osError == userCanceledErr)
        osError = noErr;

    return osError;
}

```

```
//  doCloseCommand
```

```

OSErr doCloseCommand(NavAskSaveChangesAction action)
{
    WindowPtr        windowPtr;
    SInt16           windowKind;
    docStructureHandle docStrucHdl;
    OSErr            osError = noErr;

    windowPtr = FrontWindow();
    windowKind = ((WindowPeek) windowPtr)->windowKind;

    switch(windowKind)
    {
        case kApplicationWindowKind:
            docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);
    }
}

```

```

        osError = doCloseFile(windowPtr,docStrucHdl,action);

        if(osError == kNavAskSaveChangesCancel)
            return(kNavAskSaveChangesCancel);
        else if(osError == noErr)
        {
            DisposeWindow(windowPtr);
            gCurrentNumberOfWindows --;
        }
        break;

        case kDialogWindowKind:
            // Hide or close modeless dialog, as required.
            break;
    }

    return(osError);
}

```

```

// ◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇ doSaveCommand

```

```

OSErr doSaveCommand(void)
{
    WindowPtr          windowPtr;
    docStructureHandle docStrucHdl;
    OSErr               osError = noErr;

    windowPtr = FrontWindow();
    docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);

    if((*docStrucHdl)->fileRefNum)
    {
        osError = doWriteFile(windowPtr,false);

        SetPort(windowPtr);
        EraseRect(&windowPtr->portRect);
        InvalRect(&windowPtr->portRect);
    }
    else
        osError = doSaveAsCommand();

#ifdef TARGET_CPU_PPC
    if(gMacOS85Present)
    {
        if(osError == noErr)
            SetWindowModified(windowPtr,false);
    }
#endif

    return(osError);
}

```

```

////
////
////
////
////
////

```

```

// ◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇ doSaveAsCommand

```

```

OSErr doSaveAsCommand(void)
{
    OSErr               osError = noErr;
    NavDialogOptions    dialogOptions;
    WindowPtr           windowPtr;
    docStructureHandle  docStrucHdl;
    NavEventUPP         navEventFunctionUPP;
    OSType              fileType;
    NavReplyRecord       navReplyStruc;
    AEKeyword            theKeyword;
    DescType            actualType;
    FSSpec              fileSpec;
    Size                 actualSize;
    SInt16               fileRefNum;
    Rect                portRect;

    osError = NavGetDefaultDialogOptions(&dialogOptions);

    if(osError == noErr)
    {
        windowPtr = FrontWindow();

        docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);
        GetWTitle(windowPtr,dialogOptions.savedFileName);
        GetIndString((unsigned char*) &dialogOptions.clientName,rMiscStrings,sApplicationName);
    }
}

```

```

navEventFunctionUPP = NewNavEventUPP(navEventFunction);

if((*docStrucHdl)->editStrucHdl)
    fileType = 'TEXT';
else if((*docStrucHdl)->pictureHdl)
    fileType = 'PICT';

osError = NavPutFile(NULL,&navReplyStruc,&dialogOptions,navEventFunctionUPP,
                    fileType,'kBkB',NULL);

DisposeNavEventUPP(navEventFunctionUPP);

if(navReplyStruc.validRecord && osError == noErr)
{
    if((osError = AEGGetNthPtr(&(navReplyStruc.selection),1,typeFSS,&theKeyword,
                             &actualType,&fileSpec,sizeof(fileSpec),&actualSize)) == noErr)
    {
        if(!navReplyStruc.replacing)
        {
            osError = FSpCreate(&fileSpec,'kBkB',fileType,navReplyStruc.keyScript);
            if(osError != noErr)
            {
                NavDisposeReply(&navReplyStruc);
                return(osError);
            }
        }

        (*docStrucHdl)->fileFSSpec = fileSpec;

        if((*docStrucHdl)->fileRefNum != 0)
        {
            osError = FSClose((*docStrucHdl)->fileRefNum);
            (*docStrucHdl)->fileRefNum = 0;
        }

        if(osError == noErr)
            osError = FSpOpenDF(&(*docStrucHdl)->fileFSSpec,fsRdWrPerm,&fileRefNum);

        if(osError == noErr)
        {
            (*docStrucHdl)->fileRefNum = fileRefNum;
            SetWTitle(windowPtr,fileSpec.name);
        }
    }
}

#if TARGET_CPU_PPC
    if(gMacOS85Present)
    {
        SetPort(windowPtr);
        SetWindowProxyFSSpec(windowPtr,&fileSpec);
        GetWindowProxyAlias(windowPtr,&((*docStrucHdl)->aliasHdl));
        SetWindowModified(windowPtr,false);
    }
#endif

    osError = doWriteFile(windowPtr,!navReplyStruc.replacing);
}

NavCompleteSave(&navReplyStruc,kNavTranslateInPlace);
}

NavDisposeReply(&navReplyStruc);
}

SetPort(windowPtr);
EraseRect(&windowPtr->portRect);
InvalRect(&windowPtr->portRect);
}

if(osError == userCanceledErr)
    osError = noErr;

return(osError);
}

//  doRevertCommand

OSErr doRevertCommand(void)
{
    NavEventUPP          navEventFunctionUPP;

```

```

////
////
////
////
////
////
////
////

```





```

{
    DisposeWindow(gWindowPtr);
    return(MemError());
}

SetWRefCon(gWindowPtr,(SInt32) docStrucHdl);

(*docStrucHdl)->editStrucHdl = NULL;
(*docStrucHdl)->pictureHdl   = NULL;
(*docStrucHdl)->fileRefNum   = 0;
(*docStrucHdl)->>windowTouched = false;
(*docStrucHdl)->aliasHdl     = NULL;

if(documentType == 'TEXT')
{
    gDestRect = gWindowPtr->portRect;
    InsetRect(&gDestRect,6,6);
    gViewRect = gDestRect;

    MoveHHI((Handle) docStrucHdl);
    HLock((Handle) docStrucHdl);

    if(!((*docStrucHdl)->editStrucHdl = TENew(&gDestRect,&gViewRect)))
    {
        DisposeWindow(gWindowPtr);
        DisposeHandle((Handle) docStrucHdl);
        return(MemError());
    }

    HUnlock((Handle) docStrucHdl);
}

if(showWindow)
    ShowWindow(gWindowPtr);

gCurrentNumberOfWindows ++;

return(noErr);
}

// doOpenFile
OSErr doOpenFile(FSSpec fileSpec,OSType documentType)
{
    OSErr          osError;
    SInt16         fileRefNum;
    docStructureHandle docStrucHdl;

    if(osError = doNewDocWindow(false,documentType))
        return(osError);

    SetWTitle(gWindowPtr,fileSpec.name);

    if(osError = FSpOpenDF(&fileSpec,fsRdWrPerm,&fileRefNum))
    {
        DisposeWindow(gWindowPtr);
        gCurrentNumberOfWindows --;
        return(osError);
    }

    docStrucHdl = (docStructureHandle) GetWRefCon(gWindowPtr);
    (*docStrucHdl)->fileRefNum = fileRefNum;
    (*docStrucHdl)->fileFSSpec = fileSpec;

    if(documentType == 'TEXT')
    {
        if(osError = doReadTextFile(gWindowPtr))
            return(osError);
    }
    else if(documentType == 'PICT')
    {
        if(osError = doReadPictFile(gWindowPtr))
            return(osError);
    }

#ifdef TARGET_CPU_PPC
    if(gMacOS85Present)
    {
        SetWindowProxyFSSpec(gWindowPtr,&fileSpec);
    }
#endif
}

```







```

if((*docStrucHdl)->editStrucHdl)
    fileType = 'TEXT';
else if((*docStrucHdl)->pictureHdl)
    fileType = 'PICT';

FSpCreateResFile(&(*docStrucHdl)->fileFSSpec,'kBkB',fileType,smSystemScript);

osError = ResError();
if(osError == noErr)
    fileRefNum = FSpOpenResFile(&(*docStrucHdl)->fileFSSpec,fsRdWrPerm);

if(fileRefNum > 0)
{
    osError = doCopyAResource('STR ',-16396,gAppResFileRefNum,fileRefNum);

    if(fileType == 'PICT')
    {
        doCopyAResource('pnot',128,gAppResFileRefNum,fileRefNum);
        doCopyAResource('PICT',128,gAppResFileRefNum,fileRefNum);
    }
}
else
    osError = ResError();

if(osError == noErr)
    CloseResFile(fileRefNum);

osError = ResError();
return(osError);
}

// doCopyAResource
OSErr doCopyAResource(ResType resourceType,SInt16 resourceID,SInt16 sourceFileRefNum,
                    SInt16 destFileRefNum)
{
    Handle sourceResourceHdl;
    Str255 sourceResourceName;
    ResType ignoredType;
    SInt16 ignoredID;

    UseResFile(sourceFileRefNum);

    sourceResourceHdl = GetResource(resourceType,resourceID);

    if(sourceResourceHdl != NULL)
    {
        GetResInfo(sourceResourceHdl,&ignoredID,&ignoredType,sourceResourceName);
        DetachResource(sourceResourceHdl);
        UseResFile(destFileRefNum);
        AddResource(sourceResourceHdl,resourceType,resourceID,sourceResourceName);
        if(ResError() == noErr)
            UpdateResFile(destFileRefNum);
    }

    ReleaseResource(sourceResourceHdl);

    return(ResError());
}

// navEventFunction
pascal void navEventFunction(NavEventCallbackMessage callBackSelector,
                            NavCBRecPtr callBackParms,NavCallBackUserData callBackUD)
{
    WindowPtr windowPtr;

    windowPtr = (WindowPtr) callBackParms->eventData.eventDataParms.event->message;

    switch(callBackSelector)
    {
        case kNavCBEvt:
            switch(callBackParms->eventData.eventDataParms.event->what)
            {
                case updateEvt:
                    if(((WindowPeek) windowPtr)->windowKind != kDialogWindowKind)
                        doUpdate((EventRecord *) callBackParms->eventData.eventDataParms.event);
                    break;
            }
    }
}

```

```
        break;
    }
}

// ****
// ChooseAFolderDialog.c
// ****

//
..... includes
#include "Files2.h"

// **** doChooseAFolderDialog

OSErr doChooseAFolderDialog(FSSpec *theFileSpec)
{
    NavDialogOptions dialogOptions;
    NavEventUPP      navEventFunctionUPP;
    OSErr           osError;
    NavReplyRecord  navReplyStruc;
    FSSpec         fileSpec;
    AEDesc         resultDesc;

    osError = NavGetDefaultDialogOptions(&dialogOptions);
    GetIndString(dialogOptions.message,rMiscStrings,sChooseAFolder);

    navEventFunctionUPP = NewNavEventProc((ProcPtr) navEventFunction);

    osError = NavChooseFolder(NULL,&navReplyStruc,&dialogOptions,navEventFunctionUPP,NULL,0);

    DisposeRoutineDescriptor(navEventFunctionUPP);

    if((navReplyStruc.validRecord) && (osError == noErr))
    {
        if((osError = AECOerceDesc(&(navReplyStruc.selection),typeFSS,&resultDesc)) == noErr)
        {
            BlockMoveData(*resultDesc.dataHandle,&fileSpec,sizeof(FSSpec));

            FSMakeFSSpec(fileSpec.vRefNum,fileSpec.parID,fileSpec.name,theFileSpec);
        }

        AEDisposeDesc(&resultDesc);
        NavDisposeReply(&navReplyStruc);
    }

    return(osError);
}

// ****
```

## **Demonstration Program Comments**

Note 1: Navigation Services requires Mac OS 7.5 or later and Appearance Manager 1.0.1 or later. If Mac OS 8.5 or later is not present, the Navigation Services shared library must be installed in the Extensions folder. On 680x0 systems, OpenTransportLib68K 1.3 or later must be installed in the Extensions folder.

Note 2: In addition to demonstrating Navigation Services, this program also demonstrates window proxy icons and window path pop-up menus. (See Chapter 4B - More on Windows - Mac OS 8.5 Window Manager.) Comments pertaining to these aspects of the demonstration appear on a light gray background.

When the program is run, the user should:

- Exercise the File menu by opening the supplied TEXT and PICT files, saving those files, saving those files under new names, closing files, opening the new files, attempting to open files which are already open, attempting to save files to new files with existing names, making open windows "touched" by choosing the first item in the Demonstration menu, reverting to the saved versions of files associated with "touched" windows, choosing Quit when "touched" and non-"touched" windows are open, and so on.
- Choose, via the Show pop-up menu button, the file types required to be displayed in the Open dialog.
- Choose the Choose a Folder item from the Demonstration menu to display the Choose a Folder dialog, and choose a folder using the Choose button at the bottom of the dialog. (The name of the chosen folder will be drawn in the bottom-left corner of the front window.)

- With either PICT Document or TEXT Document open:
  - With the chap16b-demo folder open so that the document's Finder icon is visible, drag the window proxy icon to the desktop or to another open folder, noting that the Finder icon moves to the latter. Then choose Touch Window from the Demonstration menu to simulate unsaved changes to the document. Note that the proxy icon changes to the disabled state. Then save the file, proving the correct operation of the file synchronisation function. Note that, after the save, the window proxy icon changes back to the enabled state.
  - Command-click the window's title to display the window path pop-up menu, choose a folder from the menu, and note that the Finder is brought to the foreground and the chosen folder opens.

The program may be run from within CodeWarrior to demonstrate responses to the File menu commands and the Choose a Folder dialog.

The built application, together with the supplied TEXT and PICT files, may be used to demonstrate the additional aspect of integrating the receipt of required Apple events with the overall file handling mechanism. To prove the correct handling of the required Apple events, the user should:

- Open the application by double-clicking the application icon, noting that a new document window is opened after the application is launched and the Open Application event is received.
- Double click on a document icon, or select one or more document icons and either drag those icons to the application icon or choose Open from the Finder's File menu, noting that the application is launched and the selected files are opened when the Open Documents event is received.
- Close all windows and double-click the application icon, noting that the application responds to the Re-open Application event by opening a new window.
- With several documents open, some with "touched" windows, choose Restart or Shut Down from the Finder's Special menu (thus invoking a Quit Application event), noting that, for "touched" windows, the Save Changes alert box is presented asking the user whether the file should be saved before the shutdown process proceeds.

## **Files2.h**

---

### **#typedef**

---

Each window created by the program will have an associated document structure, accessed via the window structure's refCon field. The docStructure structure will be used for document structures.

The editStrucHdl field will be assigned a handle to a TextEdit edit structure ('TEXT' files). The pictureHdl field will be assigned a handle to a Picture structure ('PICT' files). The fileRefNum and fileFSSpec fields will be assigned the file reference number and the file system specification structure of the file associated with the window. The windowTouched field will be set to true when a window has been made "touched", that is, when the associated document in memory has been modified by the user.

When a file is opened, the aliasHdl field will be assigned a handle to a structure of type AliasRecord, which contains the alias data for the file.

### **#define**

---

After the usual constants relating to menus, windows, and alert boxes are established, additional constants are established a 'STR#' resource containing error strings, four specific error conditions, a 'STR#' resource containing the application's name and the message string for the Choose a Folder dialog box, and the 'open' resource containing the file types list. kMaxWindows is used to limit the number of windows the user can open.

## **Files2.c**

---

Files.c is simply the basic "engine" which supports the demonstration. There is little in this file which has not featured in previous demonstration programs.

### **Global Variables**

---

gMacOS85Present will be assigned true if Mac OS 8.5 or later is present. (Because window proxy icons and window path pop-up menus are supported only under Mac OS 8.5 or later, the associated source code will be bypassed if Mac OS 8.5 or later is not present.)

gAppResFileRefNum will be assigned the file reference number of the application's resource fork.

### **main**

---

The call to Gestalt determines whether Mac OS 8.5 or later is present.

The call to NavServicesAvailable determines whether the Navigation Services shared library is installed and running on the user's system. If it is, NavLoad is called to load the library, otherwise, an error alert is presented and the program



terminates. (The call to NavLoad is optional. If the call is not made, the Navigation Services shared library will not load until your application calls a Navigation Services function, and will unload after the call completes. If the NavLoad call is made, you must call the NavUnload function before quitting so as to release the reserved memory.)

Routine descriptors for the required Apple events (less the Print Documents event) are created and a call is made to the application-defined function which installs the handlers. Also, the file reference number of the application's resource fork (which is opened automatically at application launch) is assigned to the global variable gAppResFileRefNum.

## ***EventLoop***

---

If the compilation is for PowerPC (Mac OS 8.5 does not run on 680x0 Macintoshes) and if Mac OS 8.5 or later is present, the application-defined function doSynchroniseFiles is called. (See File Synchronisation Function at Chapter 4B - More on Windows - Mac OS 8.5 Window Manager.)

## ***doInstallAEHandlers***

---

doInstallAEHandlers installs handlers for the Open Application, Re-Open Application, Open Documents, and Quit Application events. (Note that, so as to avoid the necessity to include application-defined printing functions in this program, a handler for the Print Documents event is not included in this demonstration.)

## ***doMouseDown***

---

Note that, in the inGoAway case, the constant kNavSaveChangesClosingDocument is passed in the call to doCloseCommand. This affects the text in the Save Changes alert box.

The inProxyIcon and inDrag cases recognise that a mouse-down in the proxy icon region can mean that the user wishes to drag the proxy icon (icon enabled), drag the window (icon disabled) or display the window path pop-up menu (Command key down, icon enabled or disabled).

If the proxy icon is enabled and the mouse-down is in the proxy icon, TrackWindowProxyDrag handles all aspects of the proxy icon drag process while the user drags the icon and returns noErr, in which case the local variable handled is assigned true and execution drops through to the break at the bottom of the inDrag case.

If TrackWindowProxyDrag returns errUserWantsToDragWindow, the user is either dragging the window (proxy icon is disabled) or displaying the window path pop-up menu. In this case, the local variable handled is assigned false and execution falls through to the IsWindowPathSelectClick call inside the inDrag case.

IsWindowPathSelectClick reports whether the mouse-down should activate the window path pop-up menu. If this call returns true, WindowPathSelect displays the pop-up menu, returning the item selected in the third parameter. If the menu item chosen is not the title of the window itself, the application-defined function doBringFinderToFront is called to make the Finder the frontmost process, thus ensuring that the window being opened will be visible. The local variable handled is then set to true so that DragWindow will not be called.

If IsWindowPathSelectClick returns false, DragWindow is called to take control of the dragging operation.

## ***doBringFinderToFront***

---

doBringFinderToFront is called from the inDrag case in doMouseDown. It makes the Finder the frontmost process.

The call to the application-defined function doFindProcess gets the process serial number of the Finder, which is then passed in a call to SetFrontProcess.

## ***doFindProcess***

---

doFindProcess is called by doBringFinderToFront to get the process serial number of the Finder.

The first two lines initialise the fields of a process serial number structure so that the search starts from kNoProcess. The next block initialises the fields of a process information structure which, amongst other things, contains fields for the signature (creator) and file type of the application file. (The search is for the signature (creator) 'MACS' and the type 'FNDR'.)

Within the while loop, GetNextProcess is called to get the process serial number of the next process. The call to GetProcessInformation gets information about this process into the process information structure. The processSignature and processType fields of process information structure are then examined. If they equal 'MACS' and 'FNDR' respectively, the while loop exits and the process serial number is assigned to the formal parameter outProcSerNo.

## ***doUpdate***

---

doUpdate performs such window updating as is necessary for the satisfactory execution of the demonstration aspects of the program.

## ***doMenuChoice***

---

If the Choose a Folder item in the Demonstration menu is chosen, the application-defined function which presents the Choose a Folder dialog box is called. This function returns `userCanceledErr` if the user clicked the Cancel push button in the dialog box. If an error other than `userCanceledErr` is returned, an error alert box is presented and the else block is bypassed.

The `doChooseAFolderDialog` function fills in the file system specification structure whose address is passed in its parameter. If a window is open, a rectangle in the bottom corner of the front window is erased and, if the user did not click the Cancel push button, the chosen folder's name, volume reference number, and parent directory ID are extracted from the file system specification structure and drawn in the bottom of the window.

## ***doFileMenu***

---

At the iClose case, `kNavSaveChangesClosingDocument` is passed in the call to `doCloseCommand`. This affects the wording in the Save Changes alert box. If `doCloseCommand` returns an error, and if that error is not `kNavAskSaveChangesCancel` (the user clicked the Cancel push button in the Save Changes alert box), an error alert box is presented.

At the iQuit case, `kNavSaveChangesQuittingApplication` is passed in the call to `doQuitCommand`. This affects the wording in the Save Changes alert box. If `doQuitCommand` returns an error, and if that error is not `kNavAskSaveChangesCancel` (the user clicked the Cancel push button in the Save Changes alert box), an error alert box is presented. If `kNavAskSaveChangesCancel` was not returned, `NavUnload` is called to release the memory reserved for the Navigation Services shared library, and `gDone` is set to true to cause program termination.

## ***doErrorAlert***

---

`doErrorAlert` handles errors, invoking an appropriate alert box (caution or stop) advising of the nature of the problem by error code number or straight text. Note that the program will only be terminated if the Navigation Services library is not installed and running, or in the case of the `memFullErr` error (no more space in the application heap).

## ***doTouchWindow***

---

`doTouchWindow` is called when the user chooses the Touch Window item in the Demonstration menu. Changing the content of the in-memory version of a file is only simulated in this program. The text "WINDOW TOUCHED" is drawn in window and the `windowTouched` field of the document structure is set to true.

If Mac OS 8.5 is present (meaning that window proxy icons are implemented), `SetWindowModified` is called with true passed in the modified parameter. This causes the proxy icon to appear in the disabled state, indicating that the window has unsaved changes.

## ***doSynchroniseFiles***

---

`doSynchroniseFiles` is called from the main event loop whenever a null event is received. It synchronises the file data for the application's document windows. (See File Synchronisation Function at Chapter 4B - More on Windows - Mac OS 8.5 Window Manager.)

`currentTicks` is assigned the number of ticks since system startup. As will be seen, `currentTicks` will be used to ensure that synchronisations only occur every quarter second (15 ticks). `WindowPtr` is assigned a pointer to the front non-floating window.

If 15 ticks have elapsed since the last synchronisation, the outer if block executes. The while loop walks the window list (see the call to `GetNextWindow` at the bottom of the loop) looking for associated files whose locations have changed. When the last window in the list has been examined, the loop exits.

Within the while loop, `GetWRefCon` is called to retrieve the handle to the window's document structure.

If the `aliasHdl` field of the window's document structure contains NULL, the window does not yet have a file associated with it, in which case execution falls through to the next iteration of the while loop and the next window is examined.

If the window has an associated file, the handle to the associated alias structure, which contains the alias data for the file, is retrieved. `ResolveAlias` is then called to perform a search for the target of the alias, returning the file system specification for the target file in the third parameter. After identifying the target, `ResolveAlias` compares some key information about the target with the information in the alias structure. If the information differs, `ResolveAlias` updates the alias structure to match the target and sets the `aliasChanged` parameter to true.

If the `aliasChanged` parameter is set to true, meaning that the location of the file has changed, the `fileFSSpec` field of the window's document structure is assigned the file system specification structure returned by `ResolveAlias`. Since it is also possible that the user has renamed the file, `SetWTitle` is called to set the window's title to the filename contained in the name field of the file system specification structure returned by `ResolveAlias`.

The next task is to determine whether the user has moved the file to the trash or to a folder in the trash, in which case the document must be closed.

`FindFolder` is called to get the volume reference number and parent directory ID of the trash folder.

The do/while loop walks up the parent folder hierarchy to the root folder. At the first line in the do/while loop, if the root folder has been reached (fsRtParID is the parent ID of the root directory), the file is not in the trash, in which case the loop exits at that point. At the next if statement, the volume reference number and parent directory ID of the file are compared with the volume reference number and directory ID of the trash. If they match, the file is closed, its associated memory is disposed of, and the window is disposed of.

The bottom line of the do/while loop effects the walk up the parent directory hierarchy, FSMakeFSSpec creates a file system specification structure from the current contents of the vRefNum and parID fields of newFSSpec. Since newFSSpec is also the target, the parID field is "filled in" again, at every iteration of the loop, with the parent ID of the directory passed in the second parameter of the FSMakeFSSpec call.

## ***doOpenAppEvent, doOpenDocsEvent, and doQuitAppEvent***

---

The handlers for the required Apple events are essentially identical to those in the demonstration program at Chapter 10 - Required Apple Events.

Most programs should simply open a new untitled window on receipt of an Open Application event. Accordingly, doOpenAppEvent simply calls the same function (doNewCommand) as is called when the user chooses New from the File menu.

On receipt of a Re-Open Application event, if no windows are currently open, doNewCommand is called to open a window.

The demonstration program supports both 'TEXT' and 'PICT' files. On receipt of an Open Application event, it is thus necessary to determine the type of each file specified in the event. Accordingly, within doOpenDocsEvent, the call to FSpGetFInfo returns the Finder information from the volume catalog entry for the file relating to the specified FSSpec structure. The fdType field of the FInfo structure "filled-in" by FSpGetFInfo contains the file type. This, together with the FSSpec structure, is then passed in the call to doOpenFile. (doOpenFile is also called when the user chooses Open from the File menu.)

Within the function doQuitAppEvent, the while loop entered at repeats for each open window. Within the loop, doCloseCommand is called, passing kNavSaveChangesQuittingApplication (which affects the wording in the Save Changes alert box). doCloseCommand, in turn, calls doCloseFile. doCloseFile presents the Save Changes alert box. If an error is returned by this sequence, and if the user did not click the Cancel push button in the alert box, the error handler is called. If the user clicked the Cancel button, it is necessary to interrupt the sequence of closing all open windows and re-enter the main event loop.

When the while loop eventually exits, NavUnload is called to release the memory reserved for the Navigation Services shared library, and gDone is set to true, causing the program to terminate.

## ***NewOpenCloseSave.c***

---

### ***Global Variables***

---

gWindowPtr is assigned the pointer to the graphics port of each new window as it is opened. gCurrentNumberOfWindows keeps a count of the number of windows opened. gDestRect and gViewRect are used to set the destination and view rectangles for the edit structures associated with 'TEXT' files.

### ***doNewCommand***

---

doNewCommand is the first of the file-handling functions. It is called when the user chooses New from the File menu and when an Open Application or Re-Open Application event is received.

Since this demonstration does not support the actual entry of text or the drawing of graphics, the document type passed to doNewDocWindow is immaterial. The document type 'TEXT' is passed in this instance simply to keep doNewDocWindow happy.

If the compilation is for PowerPC, if Mac OS 8.5 or later is present, and if doNewDocWindow returned no error, SetWindowProxyCreatorAndType is called to set the proxy icon for the window. (A new, untitled window, even though it has no associated file, needs a proxy icon to maintain visual consistency with other windows which have associated files.) The proxy icon will display in the disabled state, indicating, in this particular case, that the window has no associated file rather than unsaved changes.

The creator code and file type passed in the second and third parameters of SetWindowProxyCreatorAndType determine the icon to be displayed.)

### ***doOpenCommand***

---

doOpenCommand is called when the user chooses Open from the File menu.

NavGetDefaultDialogOptions initialises the specified NavDialogOptions structure with the defaults.

GetIndString assigns the application's name to the clientName field of the NavDialogOptions structure. This will then appear in the dialog box's title bar. The next line reads in the 'open' resource containing the file type list and assigns the handle to a

variable of type NavTypeListHandle.

NewNavEventUPP creates a routine descriptor for an application-defined event-handling function, which is included so as to make the Open dialog box movable and resizable.

The call to NavGetFile displays the Open dialog box. NULL in the first parameter means that the dialog box will open at the last location visited during the last call to NavGetFile. The second parameter will receive the information required by the application. Since the default options are being used, multiple file selection is allowed. The universal procedure pointer to the event-handling function is passed in the fourth parameter. No preview function is used. No filter function is used. The handle to the file type list is passed in the second last parameter.

When the Open dialog is dismissed by the user and NavGetFile returns, the first action is to dispose of the routine descriptor.

The following if block executes only if the user clicked the Open push button (or pressed the Return or Enter keys) and no error was returned. The first action is to call AECOUNTITEMS to count the number of descriptor structures in the descriptor list returned in the selection field of the NavReplyRecord structure, and which is created from FSSpec references to items selected in the Open dialog box.

The for loop repeats for each of the descriptor structures. AEGETHPTR gets the file system specification into a local variable of type FSSpec. This file system specification is then passed in the first parameter of a call to FSPGETFINFO, allowing the file type to be ascertained. The file system specification and file type are then passed in a call to the application-defined function DOOPENFILE, which creates a new window and reads in the file.

With all of the selected files read in, NavDisposeReply must be called to release the memory allotted for the NavReplyReply structure. The call to ReleaseResource frees the memory occupied by the file type list.

If the user clicks the Cancel push button in a Navigation Services dialog box, the relevant Navigation Services function returns userCanceledErr. In this event, the variable osError is set to noErr before doOpenCommand returns.

## ***doCloseCommand***

---

doCloseCommand is called when the user chooses Close from the File menu or clicks in the window's go-away box. It is also called successively for each open window when a Quit Application event is received.

The first two lines get the WindowPtr for the front window and establish whether the front window is a document window or a modeless dialog box.

If the front window is a document window, the handle to the window's document structure is retrieved from the window structure's refCon field. The WindowPtr and this handle are then passed to the application-defined function doCloseFile, together with the Navigation Services constant received in doCloseCommand's action parameter. If the window is "touched", doCloseFile presents the Save Changes alert box asking the user whether the document should be saved before it is closed. If the user clicks the Cancel push button of that alert box, doCloseFile returns kNavAskSaveChangesCancel, in which case doCloseCommand returns kNavAskSaveChangesCancel. If the user clicks either the OK or Don't Save push buttons in the alert box, and if doCloseFile returns no error, the window is closed as the final act in closing the file, and the global variable which keeps track of the number of open windows is decremented.

No modeless dialog boxes are used by this program. However, if the front window was a modeless dialog box, the appropriate action would be taken at the second case.

## ***doSaveCommand***

---

doSaveCommand is called when the user chooses Save from the File menu. It may also be called by doCloseFile if the user is attempting to close a "dirty" window.

The first two lines get the WindowPtr for the front window and retrieve the handle to that window's document structure. If a file currently exists for the document in this window, the application-defined function doWriteFile is called, otherwise the application-defined function doSaveAsCommand is called.

If the compilation is for PowerPC, and if Mac OS 8.5 or later is present, and if doSaveAsCommand returned no error, SetWindowModified is called with false passed in the modified parameter. This causes the window proxy icon to appear in the enabled state, indicating no unsaved changes.

## ***doSaveAsCommand***

---

doSaveAsCommand is called when the user chooses Save As... from the File menu. It is also called by doSaveCommand if the user chooses Save when the front window contains a document for which no file currently exists.

NavGetDefaultDialogOptions initialises the specified NavDialogOptions structure with the defaults.

The first two lines in the if block get the handle to the window's document structure, which will be required later. GetWTitle gets the window's title into the savedFileName field of a NavDialogOptions structure. This will be the default name for the saved file and will appear in the Name edit text field in the Save dialog box. The call to GetIndString copies the application's name to the clientName field of the NavDialogOptions structure. This will then appear in the dialog box's title bar.

NewNavEventUPP creates a routine descriptor for an application-defined event-handling function, which is included so as to make the Save dialog box movable and resizable. The next four lines retrieve the file type from the document structure for the front window.

The call to NavPutFile displays the Open dialog box. NULL in the first parameter means that the dialog box will open at the last location visited during the last call to NavPutFile. The second parameter will receive the information required by the application. The file type and creator are passed in the fifth and sixth parameters. When the user dismisses the dialog box, NavPutFile returns, at which point the routine descriptor for the event-handling function is disposed of.

The following if block executes only if the user clicked the Save push button (or pressed the Return or Enter keys) and no error was returned. A descriptor list is returned in the selection field of the NavReplyRecord structure. AEGGetNthPtr gets the file system specification into a local variable of type FSSpec.

If the value in the replacing field of the NavReplyRecord structure indicates that the file is not being replaced, FSpCreate is called to create a new file of the specified type and with the application's signature as the specified creator. If this call is not successful, the descriptor structure is disposed of, the NavReplyRecord structure is disposed of, and the function returns.

The file system specification structure returned by the FSpCreate call is assigned to the fileFSSpec field of the window's document structure. If a file currently exists for the document, that file is closed by the call to FSClose. The data fork of the newly created file is then opened by a call to FSpOpenDF, the fileRefNum field of the document structure is assigned the file reference number returned by FSpOpenDF, the window's title is set to the new file's name, and the application-defined function doWriteFile is called to write the document to the new file.

Just before the call to doWriteFile, if the compilation is for PowerPC, and if Mac OS 8.5 or later is present, SetWindowProxyFSSpec is called to establish a proxy icon for the window and associate the file with the window. (The creator code and file type of the file determine the icon to be displayed.) GetWindowProxyAlias assigns a copy of the alias data for the file to the aliasHdl field of the window's document structure. (This is used by the file synchronisation function.) SetWindowModified is called with false passed in the modified parameter. This causes the window proxy icon to appear in the enabled state, indicating no unsaved changes.

NavCompleteSave is called to complete the save operation. With the save completed, NavDisposeReply must be called to release the memory allotted for the NavReplyReply structure.

If the user clicks the Cancel push button in a Navigation Services dialog box, the relevant Navigation Services function returns userCanceledErr. In this event, the variable osError is set to noErr before doOpenCommand returns.

## ***doRevertCommand***

---

doRevertCommand is called when the user chooses Revert to Saved from the File menu. This function uses a Navigation Services function.

The first line creates a routine descriptor for an application-defined event handling function, which will be used so as to cause the Discard Changes alert box to be movable.

The next two lines get the WindowPtr for the front window and the handle to that window's document structure. The call to GetWTitle gets the window's title (that is, the filename). BlockMoveData copies that filename to the savedFileName field of a NavDialogOptions structure. This will be used in the Discard Changes alert box's text.

The call to NavAskDiscardChanges displays the Discard Changes alert box. The information required by the application is received in the reply parameter. When the user dismisses the alert box, the routine descriptor is disposed of.

If the user clicked the OK push button, the window's content area is erased and the appropriate application-defined function (doReadTextFile or doReadPictFile) is called depending on whether the file type is 'TEXT' or 'PICT'. In addition, the window's "touched" field in the document structure is set to false and InvalRect is called to force a redraw of the window's content region.

Just before the InvalRect call, if the compilation is for PowerPC, and if Mac OS 8.5 or later is present, SetWindowModified is called with false passed in the modified parameter. This causes the window proxy icon to appear in the enabled state, indicating no unsaved changes.

## ***doQuitCommand***

---

doQuitCommand is called when the user chooses Quit from the File menu and when a Quit Application event is received.

The while loop continues to execute until no more windows remain open. On each pass through the loop, doCloseCommand is called to manage the process of closing (and, where necessary, saving) all documents and disposing of the associated windows.

## ***doNewDocWindow***

---

doNewDocWindow is called by doNewCommand, doOpenFile and the Open Application event handler. It creates a new window and associated document structure.

If the current number of open windows is the maximum allowable by this program, the function immediately exits, passing an error code which will cause an advisory error alert box to be displayed.

The call to GetNewCWindow opens a new window. SetPort sets that window's graphics port as the current port for drawing.

The call to NewHandle allocates memory for the window's document structure. If this call is not successful, the window is disposed of and the function returns with the error code returned by MemError.

The call to SetWRefCon assigns the handle to the document structure to the window structure's refCon field. The next five lines initialise fields of the document structure.

If the document type is 'TEXT', the if block executes, creating a TextEdit edit structure and assigning a handle to that structure to the editRec field of the document structure. (Note that the processes here are not explained in detail because TextEdit and edit structures are not central to the demonstration. For the purposes of the demonstration, it is sufficient to understand that the text data retrieved from, and saved to, disk is stored in a TextEdit edit structure. TextEdit is addressed in detail at Chapter 19 — Text and TextEdit.)

If the Boolean value passed to doNewDocWindow was set to true, the call to ShowWindow makes the window visible, otherwise the window is left invisible. The penultimate line increments the global variable which keeps track of the number of open windows.

## ***doOpenFile***

---

doOpenFile is called by doOpenCommand and the Open Documents event handler, which pass to it the file system specification structure and document type. doOpenFile opens a new document window and calls the application-defined functions which read in the file.

The call to doNewDocWindow opens a new window and creates an associated document structure. SetWTitle sets the window's title. FSpOpenDF opens the file's data fork. If this call is not successful, the window is disposed of and the function returns. The next three lines assign the file reference number and file system specification structure to the relevant fields of the document structure.

The next block calls the appropriate function for reading in the file, depending on whether the file type is of type 'TEXT' or 'PICT'. If the file is read in successfully, ShowWindow makes the window visible.

Just before the call to ShowWindow, if the compilation is for PowerPC, and if Mac OS 8.5 or later is present, SetWindowProxyFSSpec is called to establish a proxy icon for the window and associate the file with the window. (The creator code and file type of the file determine the icon to be displayed.) GetWindowProxyAlias assigns a copy of the alias data for the file to the aliasHdl field of the window's document structure. (This is used by the file synchronisation function.) SetWindowModified is called with false passed in the modified parameter. This causes the window proxy icon to appear in the enabled state, indicating no unsaved changes.

## ***doCloseFile***

---

doCloseFile is called by doCloseCommand. doCloseFile does not allow a "touched" window to be closed without offering the user the option of first saving the associated document to file. This function uses a Navigation Services function.

The first if block executes only if the window has been "touched". First, GetWTitle gets the window's title, which BlockMoveData copies to the savedFileName field of a NavDialogOptions structure. This will be used in the alert box's text. A routine descriptor is created for an event-handling function, which is used so as to cause the Save Changes alert box to be movable.

NavAskSaveChanges is then called to display the Save Changes alert box. The information required by the application is received in the reply parameter. The routine descriptor is disposed of when NavAskSaveChanges returns.

If the Cancel push button was clicked, NavAskSaveChanges returns kNavAskSaveChangesCancel, in which case doCloseFile returns, returning kNavAskSaveChangesCancel to the calling function. If OK push button was clicked, the application-defined function doSaveCommand is called to save the file.

If the document has a file, FSClose closes the file, and FlushVol stores to disk all unwritten data currently in the volume buffer.

If the document is a text document, the text edit structure is disposed of. If it is a picture document, the Picture structure is disposed of. Finally, the document structure is disposed of.

## ***doWriteFile***

---

doWriteFile is called by doSaveCommand and doSaveAsCommand. In conjunction with two supporting application-defined functions, it writes the document to disk. Note that, unlike the function of the same name in the demonstration program Files1, the "safe-save" methodology is not used.

The first two lines retrieve a handle to the window's document structure and the file reference number from the document structure.

At the next four lines, the appropriate application-defined function is called to write the document's data to its file.

If the file is a newly created file, the application-defined function doCopyResources is called to copy the missing application name string resource from the resource fork of the application file to the resource fork of the new document file. If the file type is 'PICT', a 'pnot' resource and associated 'PICT' resource is also copied to the resource fork.

## ***doReadTextFile***

---

doReadTextFile is called by doOpenFile and doRevertCommand to read in data from an open file of type 'TEXT'.

The first two lines retrieve the file reference number from the document structure.

The next three lines retrieve the handle to the TextEdit edit structure from the document structure and modify the text size and line height fields of the edit structure.

SetFPos sets the file mark to the beginning of the file. GetEOF gets the number of bytes in the file. If the number of bytes exceeds that which can be stored in a TextEdit edit structure (32,767), the number of bytes which will be read from the file is restricted to 32,767.

NewHandle allocates a buffer equal to the size of the file (or 32,767 bytes if the preceding if statement executed). FSRead reads the data from the file into the buffer. MoveHHi and HLockHi move the buffer high in the heap and lock it preparatory to the call to TEsSetText. TEsSetText copies the text in the buffer into the existing hText handle of the TextEdit edit structure. The buffer is then unlocked and disposed of.

(Note: TextEdit is addressed in detail at Chapter 19 - Text and TextEdit.)

## **doReadPictFile**

doReadPictFile is called by doOpenFile and doRevertCommand to read in data from an open file of type 'PICT'.

The first two lines retrieve the file reference number from the document structure. GetEOF gets the number of bytes in the file. SetFPos sets the file mark 512 bytes (the size of a 'PICT' file's header) past the beginning of the file, and the next line subtracts the header size from the total size of the file. NewHandle allocates memory for the Picture structure and FSRead reads in the file's data.

## **doWriteTextData**

doWriteTextData is called by doWriteFile to write text data to the specified file.

The first two lines retrieve the handle to the TextEdit edit structure from the document structure. The number of bytes of text is then retrieved from the teLength field of the text edit structure.

SetFPos sets the file mark to the beginning of the file. FSWrite writes the specified number of bytes to the file. SetEOF adjusts the file's size. FlushVol stores to disk all unwritten data currently in the volume buffer.

The penultimate line sets the windowTouched field of the document structure to indicate that the document data on disk equates to the document data in memory.

## **doWritePictData**

doWritePictData is called by doWriteFile to write picture data to the specified file.

The first two lines retrieve the handle to the relevant Picture structure from the document structure. SetFPos sets the file mark to the start of the file. FSWrite writes zeros in the first 512 bytes (the size of a 'PICT' file's header). GetHandleSize gets the size of the Picture structure and FSWrite writes the bytes in the Picture structure to the file. SetEOF adjusts the file's size and FlushVol stores to disk all unwritten data currently in the volume buffer.

The penultimate line sets the windowTouched field of the document structure to indicate that the document data on disk equates to the document data in memory.

## **doCopyResources**

doCopyResources is called by doWriteFile when a newly created file has been written to for the first time. It copies the missing application name string resource from the resource fork of the application file to the resource fork of the new file. If the file type is PICT, a 'pnot' resource and associated 'PICT' resource is also copied.

The first line retrieves a handle to the file's document structure. The next four lines establish the file type involved. FSpCreateResFile creates the resource fork in the new file and FSpOpenResFile opens the resource fork. The application-defined function for copying specified resources between specified files (doCopyAResource) is then called to copy the missing application name string resource from the resource fork of the application file to the resource fork of the new file. If the file type is 'PICT', a 'pnot' resource and associated 'PICT' resource is also copied so as to provide a preview for 'PICT' files in the Open dialog box. (Of course, in a real application, the 'pnot' and 'PICT' resource would be created by the application for each separate 'PICT' file.)

CloseResFile closes the resource fork of the new file.

## **doCopyAResource**

doCopyAResource copies specified resources between specified files. In this program, it is called only by doCopyResources.

UseResFile sets the application's resource fork as the current resource file. GetResource reads the specified resource into memory.

GetResInfo, given a handle, gets the resource type, ID and name. (Note that this line is included only because of the generic nature of doCopyResource. The calling function has passed doCopyResource the type and ID in this instance.)

DetachResource removes the resource's handle from the resource map without removing the resource from memory, and converts the resource handle into a generic handle. UseResFile makes the new file's resource fork the current resource file. AddResource makes the now arbitrary data in memory into a resource, assigns a resource ID, type and name to that resource, and inserts an entry in the resource map for the current resource file. UpdateResFile then writes the resource map and data to disk.

## ***navEventFunction***

---

navEventFunction is the event-handling function used by the Navigation Services dialog and alert boxes.

The formal parameter callBackSelector is a constant indicating which type of call Navigation Services is making to navEventFunction. One such constant is kNavCBEvent, which indicates that an event has occurred. callBackParms is a pointer to a structure of type NavCBRec. The event's event structure resides in the eventData field of the NavCBRec structure.

At the first line, the window's WindowPtr is retrieved from the event structure's message field. At the kNavCBEvent case, the event type is extracted from the event structure's what field. If it is an update event, and if it is not for a Navigation Services dialog box (the application does not open any other dialog boxes), the application's window updating function doUpdate is called.

## ***ChooseAFolderDialog.c***

---

### ***doChooseAFolderDialog***

---

doChooseAFolderDialog is called when the user chooses the Choose a Folder Dialog item in the demonstration menu. This function uses Navigation Services functions.

NavGetDefaultDialogOptions initialises the specified NavDialogOptions structure with the defaults. GetIndString copies a string to the message field of a NavDialogOptions structure. This will appear immediately below the browser list in the dialog box.

The next line creates a routine descriptor for the event-handling function.

NavChooseFolder displays the Choose a Folder dialog box. When the user dismisses the dialog box, NavChooseFolder returns, at which time the routine descriptor is disposed of.

The if block executes if the user clicked the Choose push button. AECOerceDesc coerces the descriptor structure in the descriptor list returned in the selection field of the NavReplyRecord structure to a descriptor of type file system specification, the resulting descriptor being assigned to the AEDesc structure resultDesc. If this call is successful, BlockMoveData is called to copy the data from the dataHandle field of that structure to a local variable of type FSSpec.

When Navigation Services passes your application an AEDesc structure of type typeFSS describing a directory, the name field is empty and the parID field contains the directory ID of that directory, not the ID of the parent directory. In this demonstration, the volume reference number and directory ID are passed in a call to FSMakeFSSpec, which fills in the fields of the FSSpec record pointed to by the fourth parameter. In the function doMenuChoice, the contents of the fields of this FSSpec structure (the directory name, its parent directory ID, and the volume reference number) are drawn in the bottom of the front window.