18

FILES AND NAVIGATION SERVICES

Demonstration Program: Files

Introduction

This chapter addresses:

- Creating, opening, reading from, writing to, and closing **files**.
- **Navigation Services**, an application programming interface that allows your application to provide a user interface for navigating, opening, and saving Mac OS file objects.

Files

Types of Files

A file is a named, ordered sequence of bytes stored on a volume. The files associated with an application are typically:

- The **application file** itself, which comprises the application's executable code and any application-specific resources and data.
- **Document files** created by the user using the application, which the user can edit.
- A **preferences file** created by the application to store user-specified preference settings for the application.

The Operating System also uses files for certain purposes. For example, as stated at Chapter 9, the File Manager uses a special file called the volume's **catalog file** to maintain the hierarchical organisation of files and folders in a volume.

Characteristics of Files

File Forks

Macintosh files comprise two **forks**, called the **data fork** and the **resource fork**. The resource fork contains a resource map and resources. Unlike the bytes stored in the resource fork, the bytes in the data fork do not have to have any particular internal structure. Your application must therefore be able to interpret the bytes in the data fork in an appropriate manner.

All Macintosh files contain a data fork and a resource fork; however, one or both of these forks may, in fact, be empty. Fig 1 shows the typical contents of the data and resource forks of an application file and a document file.

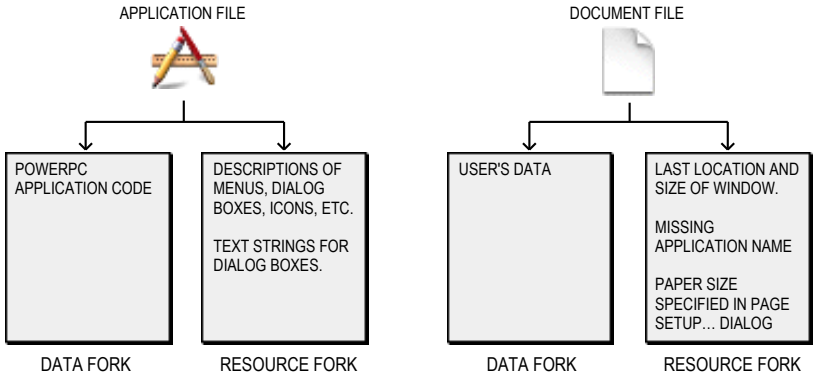


FIG 1 - TYPICAL CONTENTS OF DATA FORKS AND RESOURCE FORKS IN APPLICATION AND DOCUMENT FILES

If your data can be structured as a resource, you might elect to store that data in the resource fork, in which case you use Resource Manager functions to both store and retrieve it. Retrieving data from a resource fork is a comparatively simple matter because all you have to do is pass the resource type and ID to the relevant Resource Manager function.

If it is neither possible nor advisable to store the data in the resource fork, you must store it in the data fork. This is normally the favoured option for storing, for example, a document's text. In this case, you use File Manager functions to store and retrieve the data. With File Manager functions, unlike Resource Manager functions, you can access any byte, or group of bytes, individually.

Generally speaking, unless the data created by the user will occupy only a small number of resources, you should store it in the data fork. Always bear in mind that the Resource Manager was not designed as a general purpose data storage and retrieval system.

File Size

Volumes

A **volume**, which can be an entire disk or only part thereof, is that part of a storage device formatted to contain files. Ordinarily, file size is limited only by the size of the volume that contains it.

Logical Blocks and Allocation Blocks

Volumes are formatted into **logical blocks**. Each logical block can contain up to 512 bytes, the actual size being of interest only to the disk device driver. When the File Manager allocates space for a file, it allocates it in units called **allocation blocks**, which are groups of consecutive logical blocks. A non-empty file fork always occupies at least one allocation block.

The size of an allocation block is the chief distinguishing feature between the volume format known as the Hierarchical File System (HFS) and the newer Hierarchical File System Plus (HFS Plus or HFS+) introduced with Mac OS 8.1. The differences are as follows:

- ***HFS (Mac OS Standard Format).*** For HFS-formatted volumes, the File Manager can access a maximum of 65,535 allocation blocks on any volume. Thus the larger the volume, the larger the allocation block. For example, on a 500 MB volume, the allocation block size is 8KB under HFS.
- ***HFS Plus (Mac OS Extended Format).*** For HFS Plus-formatted volumes, the File Manager can access a maximum of 4.29 billion allocation blocks on any volume. This means that even huge volumes can be formatted with very small allocation blocks. The default volume format for Carbon is HFS Plus.

> ***Note***
>
> Beginning with Mac OS 9, HFS Plus introduced support for long Unicode filenames, files larger than 2GB, and extended file attributes. The additional File Manager constants, data types, and functions introduced at that time are often referred to as the **HFS Plus API**.

On large volumes, the significant reduction in allocation block size under HFS Plus results in significant space savings. For example, on a 4 GB volume, a file containing only 4 KB of information requires 64 KB of space under HFS, whereas the same file requires only 4KB of space under HFS Plus.

Physical and Logical End-Of-File

There is a difference between the amount of space allocated to a file and the number of bytes of actual data in the file. This is reflected in the two numbers used to describe the size of a file:

- ***Physical End-Of-File.*** The physical end-of-file is the number of bytes currently allocated to the file by the File Manager. Since the file's first byte is byte number 0, the physical end-of-file is 1 greater than the number of the last byte in its last allocation block. The physical end-of-file is thus always an exact multiple of allocation block size.
- ***Logical End-Of-File.*** The logical end-of-file is one greater than the number of bytes that currently contain data.

Fig 2 illustrates logical end-of-file and physical end-of-file.

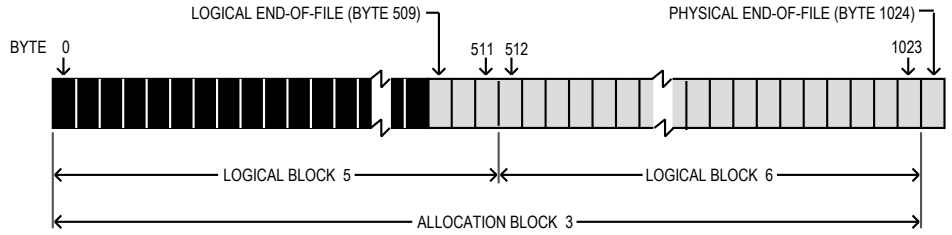


FIG 2 - LOGICAL AND PHYSICAL END-OF-FILE

Your application can adjust the size of a file by moving the logical end-of-file. If, when you increase the size of a file, the logical end-of-file is moved past the physical end-of-file, one or more allocation blocks are automatically added to the file by the File Manager. By the same token, the File Manager automatically deletes the unneeded allocation block if you move the logical end-of-file more than one allocation block short of the current physical end-of-file.

Clumps and Combating File Fragmentation

The volume's **clump** size determines the number of allocation blocks added to the file when you move the logical end-of-file past the physical end-of-file. The File Manager enlarges files by adding clumps (which are groups of *contiguous* allocation blocks) as a way of reducing file fragmentation and improving input/output performance.

Your application can also takes steps to reduce file fragmentation. Suppose you are extending a file with multiple write operations. If you know before you begin how large the file is likely to become, you should first call `SetEOF` to set the file to that size.

File Access

The operations your application can perform on a file depend on whether it is open or closed. For example, reading and writing operations can only be performed on open files, and deleting operations can only be performed on closed files.

Access Path and File Reference Number

When a file is opened, the File Manager reads in file information and creates an **access path** to the file. The file information is stored in a **file control block** (FCB). The access path, which is assigned a unique **file reference number**, specifies the volume on which the file is located and the location of the file on that volume.

File Mark

The File Manager maintains a **file mark** (a current-position marker) for each access path. The file mark, which is moved each time a byte is read or written, is the number of the next byte to be read or written. By setting the file mark or specifying an offset, you can control the beginning point of a read or write operation.

Data Buffer

When it transfers data to or from your application, the File Manager uses a **data buffer** in RAM. You must therefore pass the address of this data buffer whenever you read or write a file's data.

Disk Cache

The File Manager uses an intermediate buffer, called the **disk cache**, when reading data from, or writing data to, the file system.

During a write operation, data is transferred from your application's data buffer to the disk cache. During a read operation, the File Manager looks for data in the disk cache and, if data is found in the cache, transfers that data to your application's data buffer. If the File Manager finds no data in the disk cache, it reads the requested number of bytes from the disk directly to your application's data buffer.

The Hierarchical File System

Directories and Directory ID

The method used to organise files on a Macintosh volume is called a **hierarchical file system**. In this system, files are grouped into **directories** (also called **folders**). These directories may, in turn, be grouped into other directories (see Fig 3). As shown at Fig 3, each directory has a number associated with it called the **directory ID**.

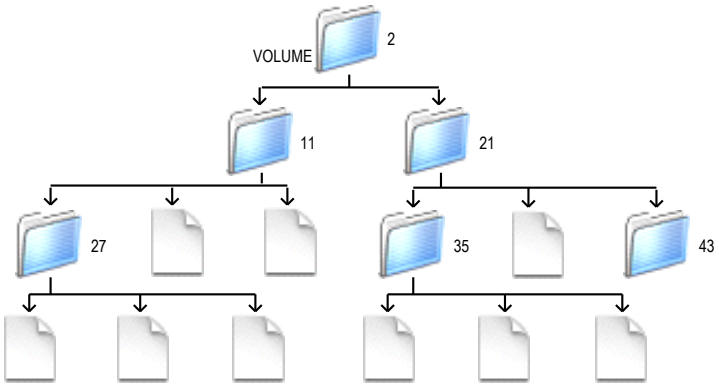


FIG 3 - MACINTOSH HIERARCHICAL FILE SYSTEM

Root Directory

The Finder and the File Manager work together to maintain the organisation of files and folders on a volume, ensuring that the representation on the desktop corresponds directly to the hierarchical directory structure on the volume. In file system parlance, the volume is referred to as the **root directory**, and the folders are referred to as **subdirectories** (or simply as **directories**).

Mounted Volumes

When a volume is **mounted**, the File Manager places information about the volume in a **volume control block** (**VCB**) and assigns a **volume reference number** by which you can refer to the volume until it is **unmounted**. Mounted volumes appear on the desktop.

You can identify a volume by its volume reference number or by its **volume name**. To avoid confusion between volumes with the same name, you should ordinarily use the volume reference number to refer to a volume.

When a volume is unmounted, the volume control block is released and the volume is no longer known to the File Manager.

Parent Directory and Parent Directory ID

The directory in which a subdirectory is located is referred to as a **parent directory**. A parent directory is assigned a **parent directory ID**. A special parent directory ID is assigned by the File Manager to a volume's root directory. All this facilitates a consistent method of identifying files and directories using the volume reference number, the parent directory ID, and the file or directory name.

Generally speaking, your application does not need to keep track of the location of files in the file system hierarchy. The location of most of the files your application opens and saves is provided by either the Finder or Navigation Services.

Aliases

An **alias** is a special kind of file that represents another file, folder, or volume. The Finder and Navigation Services automatically resolve aliases.

Identifying Files and Directories — File System Specification Structure and File System Reference

Three pieces of information are all that is needed to identify a file or directory: a volume reference number; a parent directory ID; the name of the file or directory. Of relevance is this regard are two data types, namely, the **file system specification structure** and the opaque **file system reference**:

```
struct FSSpec
{
  short vRefNum;  // Volume reference number.
  long  parID;    // Directory ID of parent directory.
  Str63 name;     // Filename or directory name.
};
typedef struct FSSpec FSSpec;
typedef FSSpec *FSSpecPtr, **FSSpecHandle;

struct FSRef
{
  UInt8 hidden[80];
};
typedef struct FSRef FSRef;
typedef FSRef *FSRefPtr;
```

The opaque data type `FSRef`, whose purpose is similar to that of the file system specification structure, was introduced with the HFS Plus API. Note that there is no need to call the File Manager to dispose of an `FSRef` when it is no longer needed.

Creating, Opening, Reading From, Writing To, and Closing Files

Your application typically creates, opens, reads from, writes to, and closes files in response to the user choosing commands from the **File** menu. In addition, your application opens, reads from, writes to, and closes files in response to the required Apple events (see Chapter 10).

The following describes how to perform typical file operations within the context of a user choosing commands from an imaginary application's **File** menu and, on Mac OS X, the **Quit** command. For the

purposes of illustration, the assumption is made that the files involved store text documents and that, when retrieved from file, the documents are displayed in a window with scroll bars.

General File Menu and Required Apple Events Handling Strategy

A suggested general strategy for handling user choices of the **New**, **Open...**, **Close**, **Save**, **Save As...**, **Revert**, and **Quit** commands, and for responding to the required Apple events, is illustrated at Fig 4.

Preliminaries - Creating a Document Structure

The contents of document files are displayed in windows. Ordinarily, your application should define a **document structure** which contains information about the window and information about the file whose contents are displayed in the window. The following is an example of a document structure for an application that handles text files:

```
typedef struct
{
  ControlHandle vScrollBarHdl;  // Handle to vertical scroll bar.
  ControlHandle hScrollBarHdl;  // Handle to horizontal scroll bar.
  SInt16        fileRefNum;     // File reference number for window's file.
  FSSpec        fileFSSpec;     // File's file system specification structure.
  TEHandle      textEditHdl;    // Handle to TextEdit structure.
  Boolean       windowTouched;  // Has window's data changed?
}  documentStructure;

typedef documentStructure *documentStructurePtr;
typedef documentStructure **documentStructureHdl;
```

Note the `fileRefNum` and `fileFSSpec` fields. Note also that the last field (`windowTouched`) is used to record whether the content of the document in memory differs from that in the associated file. Your application should set this field to `false` when it first reads in the file and immediately after each save, and to `true` when the content of the document in memory is changed by the user after the first read-in and after the subsequent saves. If the `windowTouched` field is set to `true` and the user attempts to close the document window, your application should present an alert asking the user whether the changed version of the document should be saved.

Document structures can be associated with the relevant window by storing a handle to the structure in the window object using the function `SetWRefCon`.

Creating a New Document Window

The user creates a new untitled document window using the **New...** command in the **File** menu. In addition, it is usual for an application to open a new untitled document window when it receives an Open Application Apple event from the Finder. (See `doNewCommand` at Fig 4.)

Although the function which responds to the user choosing the **New...** command and Open Application Apple event opens a new window, it should not create a new file. The reason for this is that, in the event, the user may elect not to save the document. It is thus preferable to wait until the user decides to save the new document before creating a file. Accordingly, the `fileRefNum` field of the new window's document structure should be set to `0` to indicate that no file is currently associated with this window.

Opening a File and Reading in Data

Your application will need to open a file when the user chooses the **Open...** command from the **File** menu (see `doOpenCommand` at Fig 4) and when it receives Open Documents and (on Mac OS 8/9) Print Documents Apple events.

Opening the Navigation Services Open Dialog

Your application's initial response to the user choosing the **Open...** command from the **File** menu should be to elicit a file selection from the user by creating and displaying a Navigation Services Open dialog (see Fig 6).

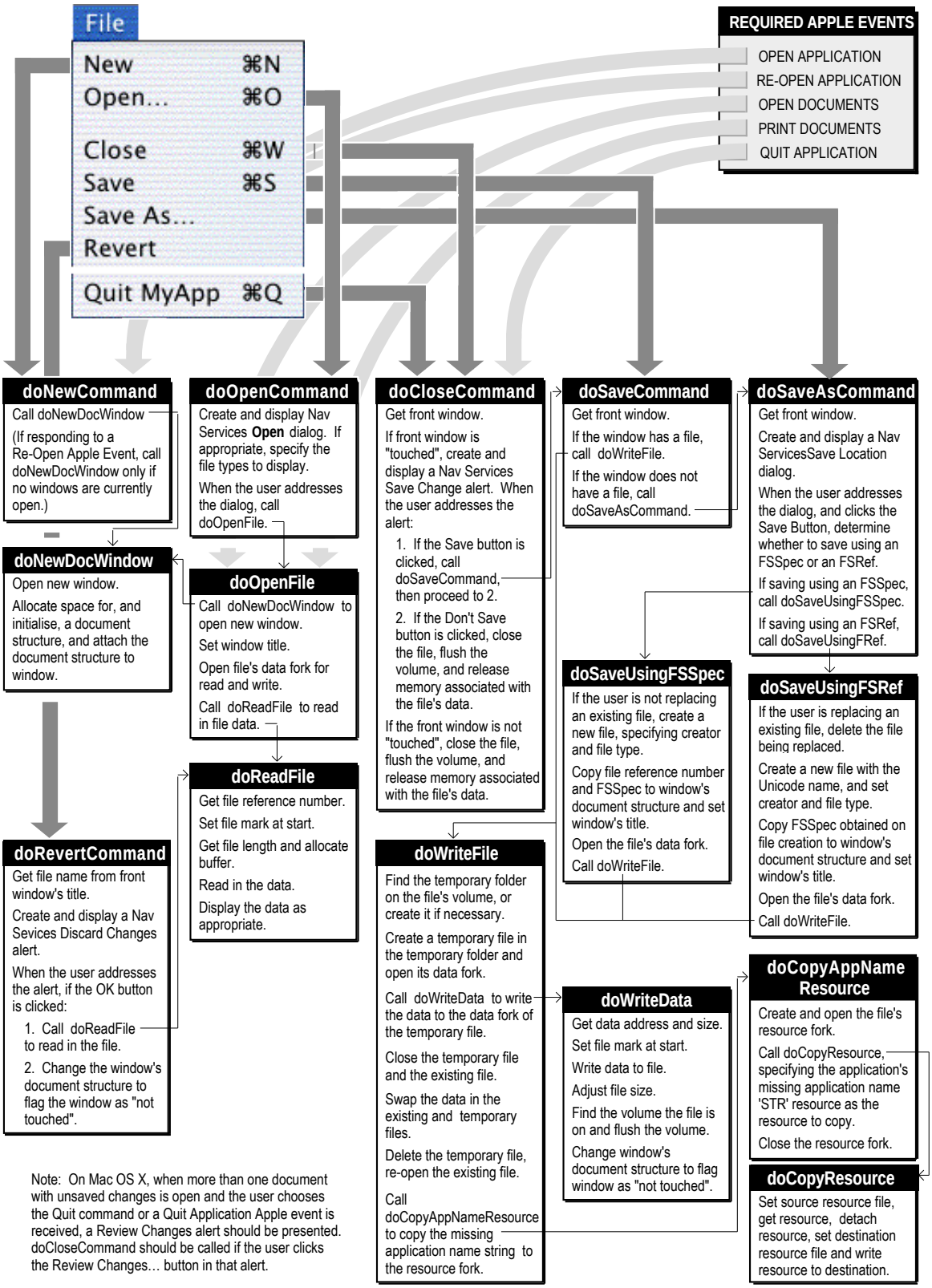


FIG 4 - GENERAL FILE MENU, QUIT ITEM, AND REQUIRED APPLE EVENTS HANDLING STRATEGY

Calls to the Navigation Services 3.0 functions `NavCreateGetFileDialog` and `NavDialogRun` create and display the Navigation Services Open dialog. When the user addresses the dialog, selects one or more files, and clicks the **Open** button, your application examines the `selection` field of a `NavReplyRecord` structure (see

Navigation Services, below) and disposes of the dialog. The `selection` field is an Apple Event descriptor list (`AEDescList`). You can determine the number of files in the list by calling the Apple Event Manager function `AECountItems`. Each selected file object is described in an `AEDesc` structure. You can coerce this descriptor into a file system specification (`FSSpec`) structure to perform operations such as opening the file.

Creating a Window and Opening the File

The next steps are to call a function (`doNewDocWindow` at Fig 4) to create a window and associated document structure and open the selected file's data fork (`doOpenFile` at Fig 4).

The file's data fork is opened using `FSpOpenDF`:

```
OSErr  FSpOpenDF(spec,permission,refNum);
FSSpec *spec;       File system specification structure.
SInt8  permission;  Access mode.
short  *refNum;     Returned file reference number.
```

`FSpOpenDF` takes the `FSSpec` returned by Navigation Services as its first parameter. The `permission` field specifies the **access mode** for opening the file. The access mode may be specified using one of the following constants:

Constant	*Value*	*Description*
`fsCurPerm`	`0`	Whatever permission is allowed.
`fsRdPerm`	`1`	Read permission.
`fsWrPerm`	`2`	Write permission.
`fsRdWrPerm`	`3`	Exclusive read/write permission.
`fsRdWrShPerm`	`4`	Shared read/write permission.

`FSpOpenDF` returns, in its third parameter, a file reference number. This reference number should be saved to the window's document structure so that it can be readily retrieved for use as a parameter in calls to functions which read from and write to the file.

Reading File Data

When you have opened a file, you can read data from it. Ordinarily, you will want to read data from the file when the user first opens it. And your application will have to read data from the file when the user chooses the **Revert** command in the **File** menu to revert to the last saved version of the document (see `doRevertCommand` at Fig 4). Typically, a function for reading file data:

- Retrieves the file reference number from the document structure.
- Calls `SetFPos` to set the file mark to the beginning of the file:

```
OSErr  SetFPos(refNum,posMode,posOff);
short refNum;   File reference number.
short posMode;  Positioning mode.
long  posOff;   Positioning offset.
```

The `posMode` parameter must contain one of the following constants:

Constant	*Value*	*Description*
`fsAtMark`	`0`	Remain at current mark.
`fsFromStart`	`1`	Set mark relative to beginning of file.
`fsFromLEOF`	`2`	Set mark relative to logical end of file.
`fsFromMark`	`3`	Set mark relative to current mark.
`rdVerify`	`64`	Add to above for read-verify.

- Determines the number of bytes in the file by calling `GetEOF`:

```
OSErr  GetEOF(refNum,logEOF);
short refNum;   File reference number.
long  *logEOF;  Receives length of file, in bytes.
```

- Calls FSRead to read the specified number of bytes from the file into the specified buffer:

```
OSErr  FSRead(refNum,count,buffPtr);
short refNum;    File reference number.
long  *count;    On input: bytes to read.  On output: actual bytes read.
void  *buffPtr;  Address of buffer into which bytes are to be read.
```

  Note that FSRead returns, in the count parameter, the actual number of bytes read.

Saving a File

The user can indicate that the current contents of a document should be saved:

- By choosing **Save** or **Save As...** from the **File** menu.
- By clicking the **Save** button in the Navigation Services Save Changes alert you present when the user attempts to close a "touched" window.
- By clicking the **Save** button in the Navigation Services Save Changes alert you present when the user attempts to quit the application while a "touched" window remains open.

Handling the Save Command

To handle the Save command (see doSaveCommand at Fig 4), your application should:

- Check the file reference number field of the window's document structure to determine if the window already has a file.
- If the window already has a file, call the function for writing files to disk (see doWriteFile at Fig 4). If the window does not have a file, call the function for handling the **Save As...** command.

Handling the Save As... Command

To handle the **Save As...** command (see doSaveAsCommand at Fig 4), your application should proceed as follows:

- Call the Navigation Services 3.0 functions NavCreatePutFileDialog and NavDialogRun to create and display the Navigation Services Save Location dialog (see Fig 10).

  When the user addresses the dialog and clicks the **Save** button, your application examines the selection field of a NavReplyRecord structure (see Navigation Services, below) and disposes of the dialog. The selection field is an Apple Event descriptor list (AEDescList). The file object is described in an AEDesc structure. If your application is running on Mac OS X, you will be able to coerce this data to type FSRef. If this coercion fails (meaning that your application is running on Mac OS 8/9) you will be able to coerce the data to type FSSpec. The FSRef or FSSpec will be required for the save operation.

- ***Save Using FSRef.*** If the coercion to type FSRef succeeds:
  - Call AEGetDescData to extract the data from the dataHandle field of the AEDesc structure. This is the FSRef for the parent directory.
  - Call CFStringGetCharacters to extract into a buffer the contents of the string referenced in the saveFileName field of the NavReplyRecord structure.
  - If the replacing field of the NavReplyRecord structure contains true, call the HFS Plus API function FSMakeFSRefUnicode to create an FSRef for the file, passing in the FSRef for the parent directory and the extracted filename characters. Pass this FSRef in a call to FSDeleteObject to delete the file:

```
OSErr  FSDeleteObject(ref);
const FSRef *ref;  Pointer to FSRef specifying file or directory to delete.
```

  - Call FSCreateFileUnicode, passing in the FSRef for the parent directory and the extracted filename characters, to create a new file with a Unicode name:

```
OSErr  FSCreateFileUnicode(parentRef,nameLength,name,whichInfo,catalogInfo,
                           newRef,newSpec);
const FSRef         *parentRef;    FSRef for directory where file to be created.
UniCharCount        nameLength;    Length of file's name.
const UniChar       *name;         Unicode name for file.
FSCatalogInfoBitmap whichInfo;     Catalog information fields to be set, if any.
const FSCatalogInfo *catalogInfo;  Values of file's catalog infoformation.
FSRef               *newRef;       On return, FSRef for new file.
FSSpec              *newSpec;      On return, FSSpec for new file.
```

- Call `FSpGetFInfo`, passing in the `FSSpec` received in the last parameter of the call to `FSCreateFileUnicode`. Assign the file type and creator to the relevant fields of the obtained `Finfo` structure and call `FSpSetFInfo` to set the Finder information.
- Assign the file system specification (`FSSpec`) structure to the file system specification structure field of the window's document structure.
- Call `FSpOpenDF` to open the data fork.
- Assign the file reference number returned by `FSpOpenDF` to the file reference number field of the window's document structure.
- Call `SetWTitle` to set the window's title, using the string extracted from the `name` field of the file system specification (`FSSpec`) structure.
- Call the function for writing files to disk (see `doWriteFile` at Fig 4).

- ***Save Using FSSpec.*** If the coercion to type `FSRef` does not succeed:
  - Call the Navigation Services 3.0 function `NavDialogGetSaveFileName` to get the file name from the edit text field of the Save Location dialog, convert it to a Pascal string using `CFStringGetPascalString`, and assign that string to the name field of the file system specification (`FSSpec`) structure.
  - If the `replacing` field of a `NavReplyRecord` structure does not contain `true`, call `FSpCreate` to create a new file and set the file type and creator:

```
OSErr  FSpCreate(spec,creator,fileType,sciptTag);
FSSpec     *spec;      File system specification structure.
OSType     creator;    File creator.
OSType     fileType;   File type.
ScriptCode scriptTag;  Code of script system in which filename is displayed.
```

  - Assign the coerced file system specification (`FSSpec`) structure to the file system specification structure field of the window's document structure.
  - If the window already has a file (that is, if the file reference number field of the document structure does not contain `0`), close that file with a call to `FSClose`:

```
OSErr  FSClose(refNum);
short refNum;  File reference number.
```

  - Call `FSpOpenDF` to open the data fork.
  - Assign the file reference number returned by `FSpOpenDF` to the file reference number field of the window's document structure.
  - Call `SetWTitle` to set the window's title, using the string extracted from the `name` field of the file system specification (`FSSpec`) structure.
  - Call the function for writing files to disk (see `doWriteFile` at Fig 4).

Writing File Data

The function for writing data (see `doWriteFile` at Fig 4) should write to a temporary file, not to the document file itself. If you write directly to the document's file, you risk corrupting that file if the write

operation does not complete successfully. The broad approach for saving data *safely* to disk is therefore to write the data to a temporary file and then, assuming the temporary file has been written successfully, swap the contents of the temporary file and the document's file.

The procedure for updating a file safely is as follows:

- Get the file system specification from the document structure.
- Create a temporary filename for the temporary file.
- Call `FindFolder` to find the temporary folder on the file's volume, or create it if necessary:

```
OSErr  FindFolder(vRefNum,folderType,createFolder,foundVRefNum,foundDirID);
short   vRefNum;        Volume reference number.
OSType  folderType;     Folder type for volume.
Boolean createFolder;   kCreateFolder or kDontCreateFolder.
short   *foundVRefNum;  Volume reference number for folder found.
long    *foundDirID;    Directory ID of folder found.
```

- Call `FSMakeFSSpec` to make a file system specification structure for the temporary file:

```
OSErr  FSMakeFSSpec(vRefNum,dirID,fileName,spec);
short            vRefNum;   Volume reference number.
long             dirID;     Parent directory ID.
ConstStr255Param fileName;  Full or partial pathname.
FSSpec           spec;      Pointer to FSSpec structure.
```

- Call `FSpCreate` to create the temporary file, and `FSpOpenDF` to open the temporary file's data fork.
- Call the function for writing data to a file (see `doWriteData` at Fig 4). This function should:
  - Retrieve the address and length of the buffer (for example, from a TextEdit structure).
  - Call `SetFPos` to set the file mark to the beginning of the file.
  - Call `FSWrite` to write the buffer to the file:

```
OSErr  FSWrite(refNum,count,buffPtr);
short      refNum;    File reference number.
long       *count;    On input: bytes to write.  On output: bytes written.
const void *buffPtr;  Address of buffer containing data to write.
```

  - Call `SetEOF` to resize the file to the number of bytes actually written:

```
OSErr  SetEOF(refNum,logEOF);
short refNum;  File reference number.
long  logEOF;  Logical end-of-file.
```

  - Call `GetVRefNum` to determine the volume containing the file:

```
OSErr  GetVRefNum(refNum,vRefNum);
short refNum;    File reference number.
short *vRefNum;  Receives volume reference number.
```

  - Call `FlushVol` to flush the volume:

```
OSErr  FlushVol(volName,vRefNum);
ConstStr63Param volName;  Pointer to name of mounted volume
short           vRefNum;  Volume reference number.
```

    Flushing the volume ensures that both the file's data and the file's catalog entry[1] are updated.
- Call `FSClose` to close the temporary file.
- Call `FSClose` to close the existing file.

[1] The catalog entry for a file contains fields that describe the physical data (such as the first allocation block and the physical and logical ends of both the resource and data forks) and fields that describe the file within the file system, such as file ID and parent directory ID.

- Call `FSpExchangeFiles` to swap the contents of the temporary file and the existing file:

```
OSErr  FSpExchangeFiles(source,dest);
const FSSpec *source;  Source file.
const FSSpec *dest;    Destination file.
```

  `FSpExchangeFiles` does not actually move the data on the volume. It merely changes the information in the volume's catalog file and, if the files are open, their file control blocks (FCBs).

- Call `FSpDelete` to delete the temporary file:

```
OSErr  FSpDelete(spec);
const FSSpec *spec;  File system specification.
```

- Call `FSpOpenDF` to re-open the data fork of the existing file.

As a final step for Mac OS 8/9, you should call a function which copies the missing application name string resource (see Chapter 9) from the resource fork of the application file to the resource fork of the newly created file. This function (`doCopyAppNameResource` at Fig 4) should:

- Call `FSpCreateResFile` to create the new file's resource fork:

```
void  FSpCreateResFile(spec,creator,fileType,sciptTag);
const FSSpec *spec;      File system specification structure.
OSType       creator;    File creator.
OSType       fileType;   File type.
ScriptCode   scriptTag;  Code of script system.
```

- Call `FSpOpenResFile` to open the resource fork:

```
short  FSpOpenResFile(spec,permission);
const FSSpec *spec;       File system specification structure.
SignedByte   permission;  Permission code.
```

  The constants used to specify the access mode in the `FSpOpenDF` call (see above) are also used to specify the permission code in the `FSpOpenResFile` call.

- Call a function (`doCopyResource` at Fig 4), which copies specified resources from one resource file to another, to copy the missing-application name `'STR '` resource (ID -16396) from your application's resource fork to the resource fork of the newly-created file.

- Call `FSClose` to close the resource fork.

Reverting to a Saved File

To allow the user to revert to the last saved version of a document, your application can include a **Revert** command in the **File** menu. To handle this command (see `doRevertCommand` at Fig 4), you should call the Navigation Services 3.0 functions `NavCreateAskDiscardChangesDialog` and `NavDialogRun` to create and display a Navigation Services Discard Changes alert (see Fig 13). When the user addresses the dialog, and clicks on the **OK** button, you simply call your function for reading file data (`doReadFile` at Fig 4) to read the file back into the window.

Closing a File

Your application should ordinarily close a file when the user clicks in the close box of the associated window or chooses the **Close** command from the **File** menu. You may also need to close files when the user chooses **Quit** from the **File** menu or a Quit Application Apple event is received from the Finder.

When your application needs to close a file, it should first check whether the associated window has been "touched" (see `doCloseCommand` at Fig 4). If the window has been "touched", you should call the Navigation Services 3.0 functions `NavCreateAskSaveChangesDialog` and `NavDialogRun` to create and display a Navigation Services Save Changes alert (see Fig 12). When the user addresses the dialog:

- If the **Save** button is clicked, call the function for saving files (`doSaveCommand` at Fig 4), call `FSClose` to close the file, call `FlushVol` to ensure that both the file's data and the file's catalog entry are updated,

set the file reference number field in the document structure to `0`, and release memory associated with the storage of the file's data. Then dispose of the document structure and, finally, the window.

- If the **Don't Save** button is clicked, perform the same actions as are performed when the **Save** button is clicked except for the call to the function for saving files.

If the window has not been "touched", perform the same actions as are performed when the **Save** button is clicked in a Save Changes alert except for the call to the function for saving files.

File Synchronisation Functions

It is always possible that, while a document file is open, the user may drag its Finder icon or proxy icon to another folder (including the Trash) or change the name of the file via the Finder icon. The application itself has no way of knowing that this has happened and will assume, unless it is informed otherwise, that the document's file is still at its original location with its original name. For this reason, applications often include a frequently-called **file synchronisation function** which synchronises the application with the actual current location (and name) of its currently open document files.

In applications which use the Classic event model, file synchronisation functions should be called after every call to `WaitNextEvent`. In applications that use the Carbon event model, a timer should be installed to trigger repeated calls to the file synchronisation function. For each of the application's document windows, the synchronisation function should update the application's internal data structures to match that of the document file as it exists on disk. The function should also ensure that, where necessary, the name of the document window is changed to match the current name of the document file on disk and close the document window if the document file has been moved to the Trash folder.

Navigation Services

The user interface for opening and saving files, confirming saves and discarding changes, choosing a volume, folder, file, or file object, creating a new folder, file format translation, and easy navigation of the file system is provided by Navigation Services.

The following reflects Navigation Service 3.0, which was introduced with CarbonLib 1.1, and which established a new model for the creation, display, and handling of Navigation Services dialogs and alerts. Navigation Services 3.0 also introduced support for Unicode and, on Mac OS X, support for sheets and the ability to specify the modality of a dialog.

Navigation Services Dialogs and Alerts

The primary dialogs created by Navigation Services are as follows:

- Open.
- Save Location.
- Choose a Volume.
- Choose a Folder.
- Choose a File.
- Choose a File Object.

The primary alerts created by Navigation Services are as follows:

- Save Changes
- Discard Changes.

A further alert, which is applicable only on Mac OS X, and for which no Navigation Services creation function existed at the time of the first release of Mac OS X, is the Review Changes alert.

The secondary dialogs and alerts created by Navigation Services are as follows:

- New Folder dialog.
- Replace Confirmation alert.
- Mac OS 8/9 Stationery option dialog.

Modality

On Mac OS 8/9, all primary Navigation Services dialogs are movable modal provided an application-defined event handling (callback) function is provided.

On Mac OS X, your application should ensure that:

- The Save Location dialog, Save Changes alert, and Discard Changes alert are window-modal (that is, sheets).
- The other primary dialogs are application-modal.

Standard User Interface Elements in Primary Dialogs

The standard user interface elements in Navigation Services primary dialogs are shown at Fig 5.

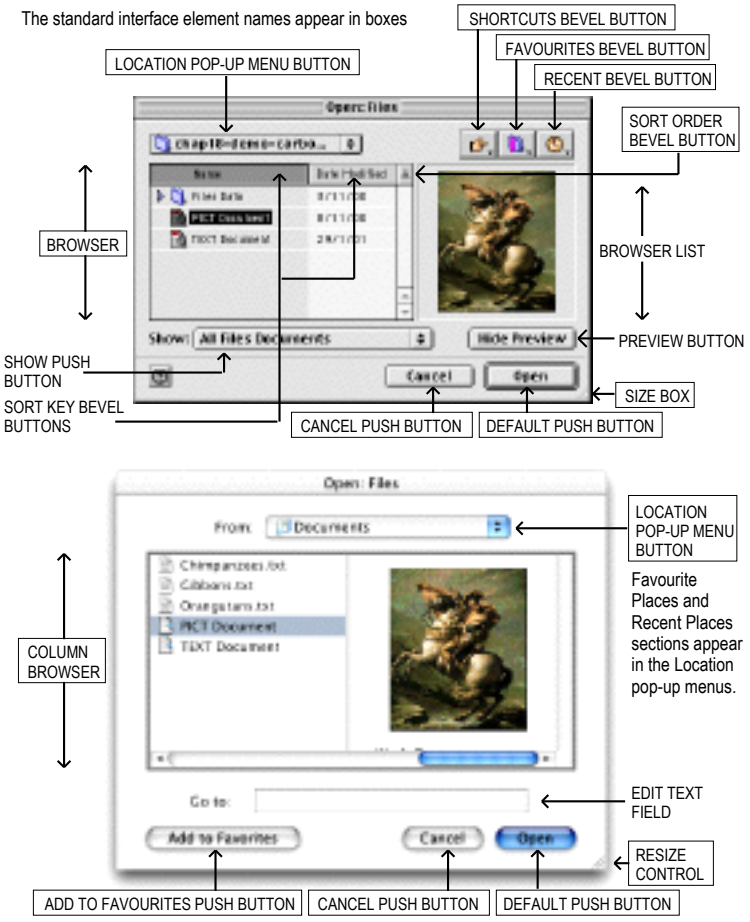


FIG 5 - STANDARD USER INTERFACE ELEMENTS IN NAVIGATION SERVICES DIALOGS

Preview

On Mac OS 8/9, the user can toggle a preview area on an off using the **Show/Hide Preview** push button in the Open dialog. A preview of any file that contains a valid `'pnot'` resource will be displayed in this area.

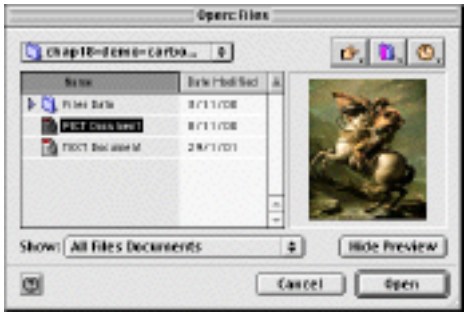
On Mac OS X, the preview appears in the column browser as shown at Fig 5. For files of type `'TEXT'` a preview is automatically created.

Persistence

Navigation Services has the ability to store information, and to store it on a per-application basis. This ability is called **persistence**. For example, when a primary dialog is displayed, the browser defaults to the directory location that was in use when that particular dialog was last closed by that particular application. In addition, if a file or folder was selected when the dialog was last closed, that file or folder is automatically selected when the dialog is re-opened. For dialogs that are not sheets, the size, position, and, on Mac OS 8/9, sort key and sort order are also remembered for each application.

Creating and Displaying an Open Dialog

The Open dialog (see Fig 6) is created by a call to `NavCreateGetFileDialog` and displayed by a call to `NavDialogRun`. You pass a universal procedure pointer to an application-defined event handling (callback) function in the `inEventProc` parameter of `NavCreateGetFileDialog`.



Note: The Show pop-up menu button appears only if a handle to a NavTypeList structure is passed in the inTypeList parameter of the NavCreateGetFileDialog function

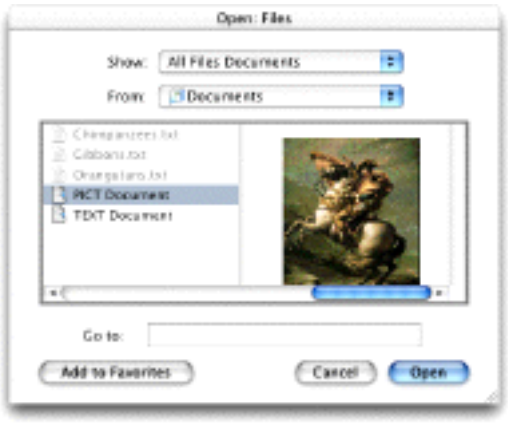


FIG 6 - NAVIGATION SERVICES OPEN DIALOG

The NavDialogCreationOptions Structure

You pass a pointer to a `NavDialogCreationOptions` structure, which specifies options controlling the appearance and behaviour of the dialog, in the `inOptions` parameter of `NavCreateGetFileDialog`. The `NavDialogCreationOptions` structure is as follows:

```
struct NavDialogCreationOptions
{
  UInt16              version;
  NavDialogOptionFlags optionFlags;
  Point               location;
  CFStringRef         clientName;
  CFStringRef         windowTitle;
  CFStringRef         actionButtonLabel;
  CFStringRef         cancelButtonLabel;
  CFStringRef         saveFileName;
  CFStringRef         message;
  UInt32              preferenceKey;
  CFArrayRef          popupExtension;
  WindowModality      modality;
  WindowRef           parentWindow;
  char                reserved[16];
};
typedef struct NavDialogCreationOptions NavDialogCreationOptions;
```

Field Descriptions

`optionsFlags` One of the following constants of type `NavDialogOptionFlags`:

Constant	*Description*

`kNavDefaultNavDlogOptions`	Use default options. Sets the following bits:
	`kNavDontAddTranslateItems`
	`kNavAllowStationery`
	`kNavAllowPreviews`
	`kNavAllowMultipleFiles`

`kNavNoTypePopup`	Don't show file type pop-up menu button.
`kNavDontAutoTranslate`	Don't auto-translate files. (Application will translate.)
`kNavDontAddTranslateItems`	Don't add translation options in Show pop-up menu.
`kNavAllFilesInPopup`	Add **All Documents** menu item in file type pop-up.
`kNavAllowStationery`	Allow stationery files.
`kNavAllowPreviews`	Allow preview to show.
`kNavAllowMultipleFiles`	Allow multiple items to be selected.
`kNavAllowInvisibleFiles`	Allow invisible items to be shown.
`kNavDontResolveAliases`	Don't resolve aliases.
`kNavSelectDefaultLocation`	Make the default location the browser selection.
`kNavSelectAllReadableItem`	Make dialog select **All Readable Documents** on open.
`kNavSupportPackages`	Recognise file system packages.
`kNavAllowOpenPackages`	Allow opening of packages.
`kNavDontAddRecents`	Don't add chosen objects to Recents list.
`kNavDontUseCustomFrame`	Don't draw the custom area bevel frame.
`kNavDontConfirmReplacement`	Don't show the "Replace File?" alert on save conflict.

`location` Location of the upper-left of the dialog (global coordinates). The dialog will appear at the location at which it was last closed if the `optionFlags` field is assigned `NULL` or this field is assigned (`-1,-1`).

`clientName` Name of your application. This will appear in the dialog's title bar.

`windowTitle` Overrides the default window title.

`actionButtonLabel` Title for the dialog's OK push button. If a title is not assigned, the push button will use the default title (**Open** or **Save**).

`cancelButtonLabel` Title for the dialog's Cancel push button. If a title is not assigned, the push button will use the default title (**Cancel**).

`savedFileName` Default file name for a saved file.

`message` A string, which is displayed in the dialog, providing additional instructions to the user. (For an example, see Fig 11).

`preferenceKey` A value that identifies the set of dialog preferences that should be used. Assign `0` if you do not wish to provide a preference key.

`popupExtension` A handle to one or more structures of type `NavMenuItemSpec` used to add extra menu items to an Open dialog's **Show** pop-up menu or a Save Location dialog's **Format** pop-up menu.

`modality` The dialog's modality (relevant on Mac OS X only). Relevant Window Manager constants are:

```
kWindowModalityNone
kWindowModalitySystemModal
kWindowModalityAppModal
kWindowModalityWindowModal
```

`parentWindow` The dialog's parent window. (Relevant on Mac OS X only when the dialog is window-modal.)

The function `NavGetDefaultDialogCreationOptions` may be called to initialise a structure of type `NavDialogCreationOptions` with the default dialog options, which are as follows:

- **Show** and **Format** pop-up menu buttons are displayed in the Open and Save Location dialogs (Mac OS 8/9).
- Files are auto-translated. (This implies that the application will not translate.)
- Translation options are not included in the **Show** pop-up menu in the Open dialog.
- The **All Documents** item is not included in the **Show** pop-up menu in the Open dialog.
- The **Stationery Option...** item is included in the **Format** pop-up menu in the Save Location dialog.
- Previews of selected files, when available, are displayed in the Open dialog.
- Selection of multiple files in the browser list/column browser in the Open dialog is allowed.
- Invisible files are nor displayed.
- Aliases are not resolved.
- The default location in the browser list/column browser is not selected.
- The **All Readable Documents** is not made the default selection in the **Show** pop-up menu in the Open dialog.
- File system packages are not displayed.
- File system packages cannot be opened and navigated.
- Chosen file objects are added to the Recents list.
- A border is drawn around the custom area.
- The default titles for the OK and Cancel buttons are used.
- No message is displayed in the dialog.

The Show Pop-up Menu

The types of files to be displayed in the browser list may be chosen from a list of available **file types** in the **Show** pop-up menu in the Open dialog (see Fig 7). This list is built from information supplied by your application in a structure of type `NavTypeList` (see below), a handle to which you pass in the `inTypeList` parameter of `NavCreateGetFileDialog`.

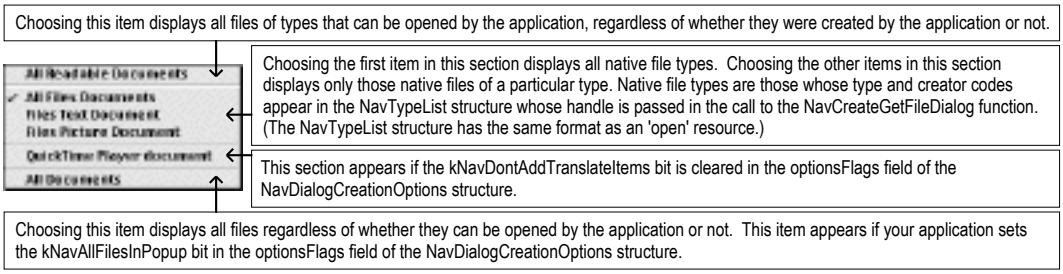


FIG 7 - THE SHOW POP-UP MENU AND FILE TYPE OPTIONS (MAC OS 8/9)

The **Show** pop-up menu button will not appear in the Open dialog if you pass `NULL` in the `inTypeList` parameter of the `NavCreateGetFileDialog` function or if you set the `kNavNoTypePopup` bit in the `optionsFlags` field of the `NavDialogCreationOptions` structure passed in the call to `NavCreateGetFileDialog`.

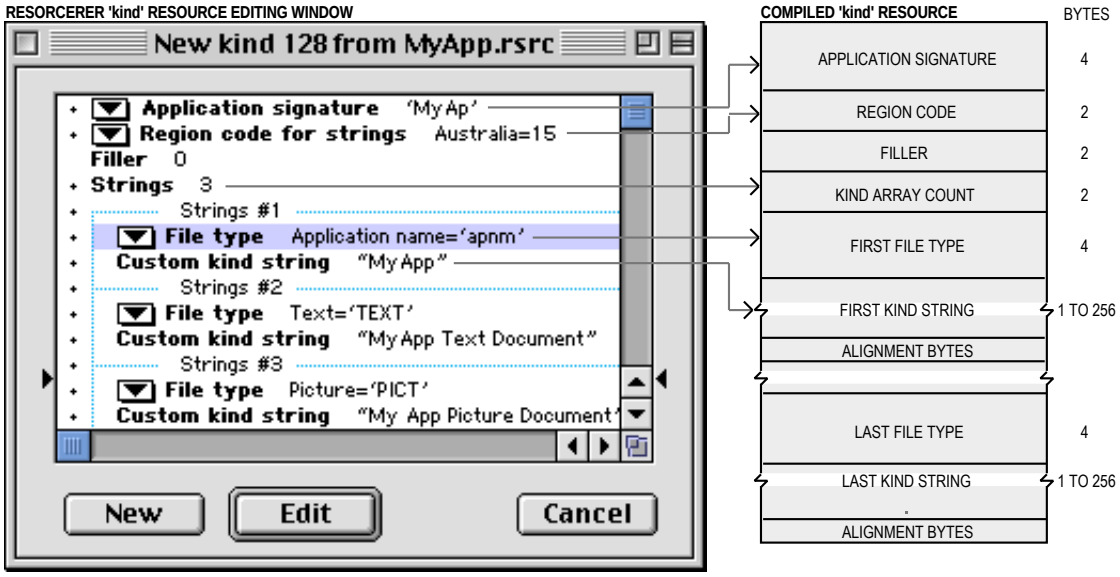
If a handle to a `NavTypeList` structure is passed in the `inTypeList` parameter and the `kNavNoTypePopup` bit is set:

- All items in the browser will be deactivated except for the file types specified in the `NavTypeList` structure whether they were created by the application or not.
- The **Show** pop-up menu button will not appear.

Native File Types Section

The first item in the **native file types** section of the Mac OS 8/9 **Show** pop-up menu defaults to **All Known Documents** if you do not assign the name of your application to the `clientName` field of the `NavDialogCreationOptions` structure passed in the `dialogOptions` parameter of the `NavCreateGetFileDialog` 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 (see below). For Mac OS 8/9, you can do this by including a **kind resource** (a resource of type `'kind'`) in your application's resource fork. Fig 8 shows the structure of a compiled `'kind'` resource and such a resource being created using Resorcerer.[2]



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

FIG 8 - STRUCTURE OF A COMPILED 'kind' RESOURCE AND CREATING A 'kind' RESOURCE USING RESORCERER

For Mac OS X, the `'kind'` resource is ignored if you provide necessary information in your application's `'plst'` resource. The relevant keys are `CFBundleDevelopmentRegion`, `CFBundleSignature`, and `CFBundleDocumentTypes`. `'apnm'` as a `CFBundleTypeOSTypes` has same effect as in `'kind'` resource.

The NavTypeList Structure

The `NavTypeList` structure, which defines the list of file types that your application is capable of opening, 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.
};
```

[2] The kind strings from your application's `'kind'` resource also appear in the Kind column in Finder window list views.

```
typedef struct NavTypeList NavTypeList;
typedef NavTypeList *NavTypeListPtr;
typedef NavTypeListPtr *NavTypeListHandle;
```

You can create your file type list dynamically or you can use an `'open'` resource. Fig 9 shows the structure of a compiled `'open'` resource and such a resource being created using Resorcerer.

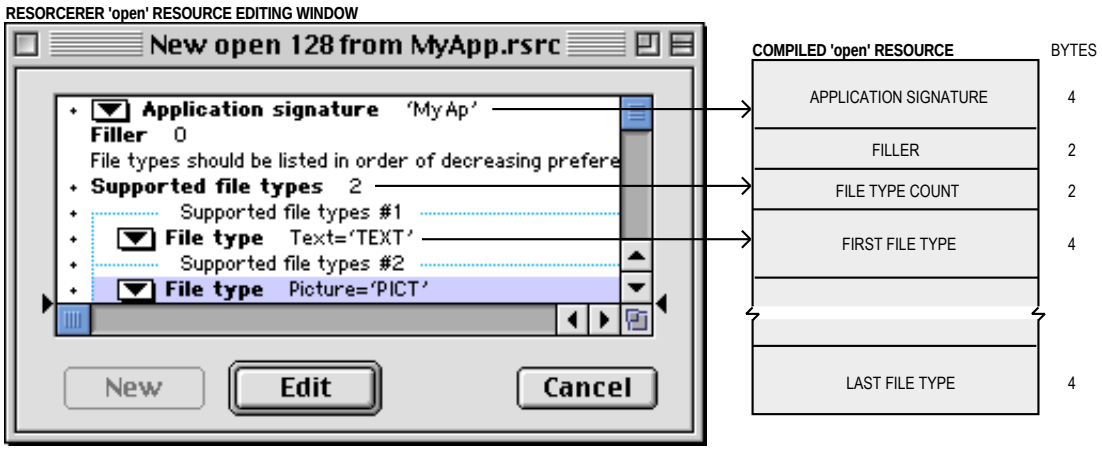
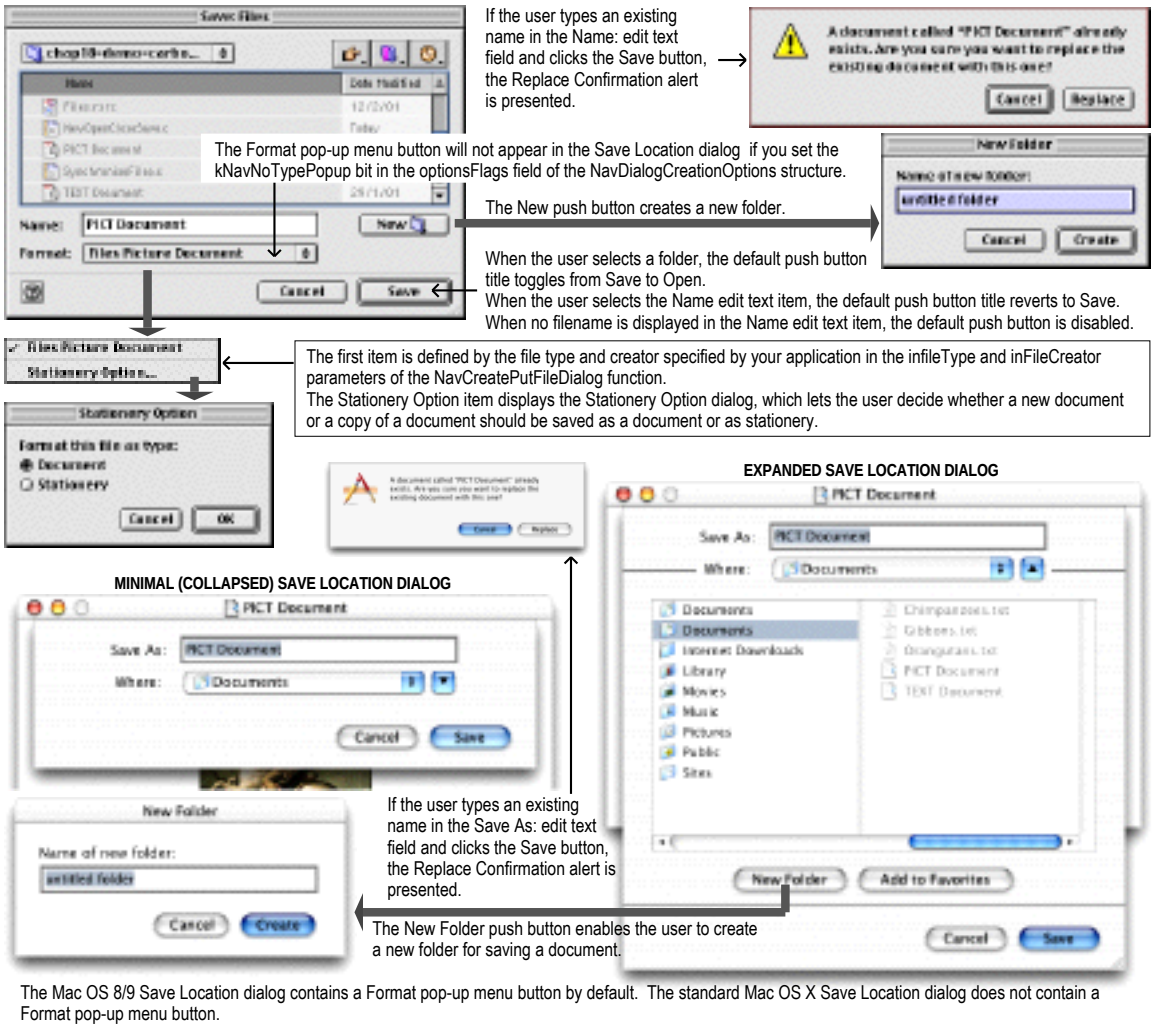


FIG 9 - STRUCTURE OF A COMPILED 'open' RESOURCE AND CREATING AN 'open' RESOURCE USING RESORCERER

Creating and Displaying a Save Location Dialog

The Save Location dialog (see Fig 10) is created by a call to `NavCreatePutFileDialog` and displayed by a call to `NavDialogRun`. You pass a universal procedure pointer to an application-defined event handling (callback) function in the `inEventProc` parameter of `NavCreatePutFileDialog`.

As with `NavCreateGetFileDialog`, you pass a pointer to a `NavDialogCreationOptions` structure in the `inOptions` parameter of `NavCreatePutFileDialog`. Other parameters allow you to specify file type and file creator.



The Mac OS 8/9 Save Location dialog contains a Format pop-up menu button by default. The standard Mac OS X Save Location dialog does not contain a Format pop-up menu button.

FIG 10 - THE SAVE LOCATION DIALOG BOX (PARTIAL) AND ASSOCIATED DIALOGS AND ALERTS

Creating and Displaying a Choose a Folder Dialog

The Choose a Folder dialog (see Fig 11) is created by a call to `NavCreateChooseFolderDialog` and displayed by a call to `NavDialogRun`. You pass a universal procedure pointer to an application-defined event handling (callback) function in the `inEventProc` parameter of `NavCreateChooseFolderDialog` and a pointer to a `NavDialogCreationOptions` structure in the `inOptions` parameter.

The other dialogs in the Choose family are created and displayed in a similar manner:

- The Choose a Volume dialog is created by a call to `NavCreateChooseVolumeDialog`.
- The Choose a File dialog is created by a call to `NavCreateChooseFileDialog`, and may be used when you want the user to select a file for a purpose other than opening. The file could be, for example, a preferences file or a dictionary file.
- The Choose a File Object dialog is created by a call to `NavCreateChooseObjectDialog`, and may be used when you need the user to select an object that might be one of several different types.

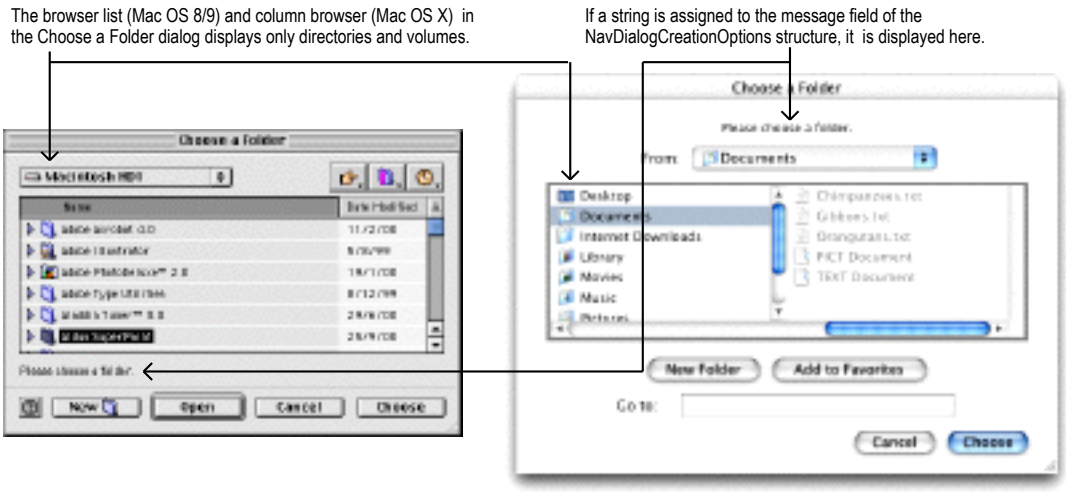


FIG 11 - CHOOSE A FOLDER DIALOG

Creating and Displaying Primary Alerts

Save Changes Alert

The Save Changes alert (see Fig 12) is created by a call to `NavCreateAskSaveChangesDialog` and displayed by a call to `NavDialogRun`. You pass a universal procedure pointer to an application-defined event handling (callback) function in the `inEventProc` parameter of `NavCreateAskSaveChangesDialog` and a pointer to a `NavDialogCreationOptions` structure in the `inOptions` parameter.

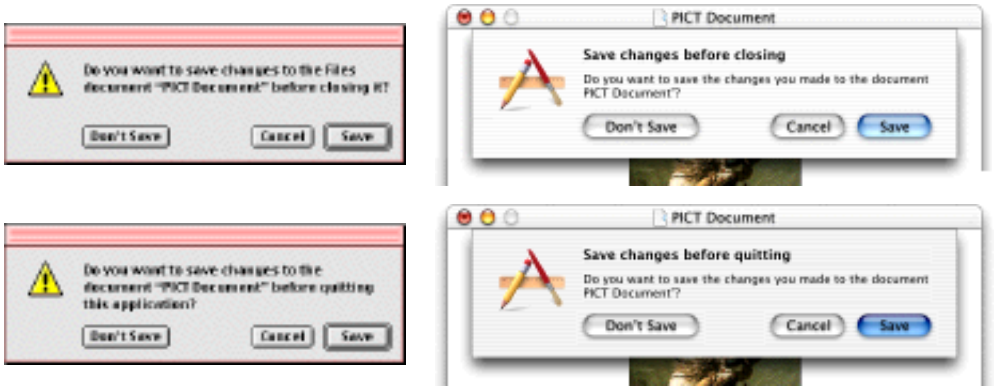


FIG 12 - SAVE CHANGES ALERTS (CLOSING DOCUMENT AND QUITTING APPLICATION)

One of the following constants is passed in the `inAction` parameter of the `NavCreateAskSaveChangesDialog` function:

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

Discard Changes Alert

To support the **Revert** command in your application's **File** menu, Navigation Services provides the Discard Changes alert. The Discard Changes alert (see Fig 13) is created by a call to `NavCreateAskDiscardChangesDialog` and displayed by a call to `NavDialogRun`. You pass a universal procedure pointer to an application-defined event handling (callback) function in the `inEventProc` parameter of `NavCreateAskDiscardChangesDialog` and a pointer to a `NavDialogCreationOptions` structure in the `inOptions` parameter.

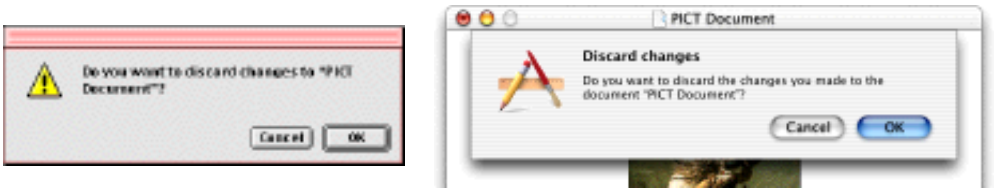


FIG 13 - DISCARD CHANGES ALERT

Review Changes Alert — Mac OS X

On Mac OS X, when the user attempts to quit your application when there is more than one document with unsaved changes open, your application should present a Review Changes alert (see Fig 14). No Navigation Services creation function existed at the time of writing; accordingly, at the time of writing, it was necessary to create, display, and handle this alert using `StandardAlert` or `CreateStandardAlert`.

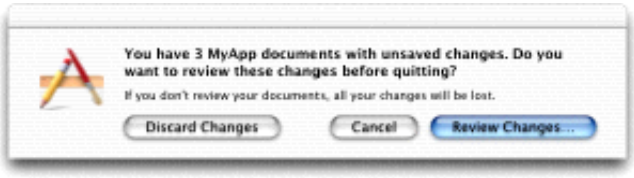


FIG 14 - REVIEW UNSAVED ALERT

A click on the **Discard Changes** button should cause all windows to close (without saving changes) and the application to close down. A click on the **Cancel** button should cancel the Quit command, keeping the application running. A click on the **Review Changes...** button should cause each window with unsaved changes to be sequentially presented to the user with a Save Changes alert presented.

Event Handling in the Primary Dialogs

Event-Handling Function

As previously stated, you pass a universal procedure pointer to an application-defined event handling (callback) function in the `inEventProc` parameter of those functions which create the Navigation Services primary dialogs. For an event handling function named `myNavEventFunction`, you would declare the function as follows:

```
void  myNavEventFunction(NavEventCallbackMessage callBackSelector,
                         NavCBRecPtr callBackParms,NavCallBackUserData callBackUD)
```

`callBackSelector` The type of event, as represented by an event message constant. Typical event message constants and their meanings are as follows:

Constant	*Value*	*Description*
`kNavCBUserAction`	`12`	The user has taken an action such as clicking on an **Open** or **Save** button.
`kNavCBEvent`	`0`	An event has occurred. Receipt of this event type allows your handler to update other windows, track controls, etc.
`kNavCBTerminate`	`3`	The dialog is about to be closed.

`callBackParms` A pointer to a `NavCBRec` structure, which contains data used by your application to process the event:

```
struct NavCBRec
{
  UInt16        version;
  NavDialogRef  context;
  WindowRef     window;
  Rect          customRect;
  Rect          previewRect;
  NavEventData  eventData;
  NavUserAction userAction;
  char          reserved[218];
};
typedef struct NavCBRec NavCBRec;
typedef NavCBRec *NavCBRecPtr;
```

`callBackUD` A pointer to a value passed in the `inClientData` parameter of the dialog creation functions.

kNavCBUserAction Message Received

When the `kNavCBUserAction` message is received, your application typically calls `NavDialogGetReply` to obtain the results of the dialog session, which are returned in a `NavReplyRecord` structure.

The NavReplyRecord Structure

```
struct NavReplyRecord
{
  UInt16                          version;
  Boolean                         validRecord;
  Boolean                         replacing;
  Boolean                         isStationery;
  Boolean                         translationNeeded;
  AEDescList                      selection;
  ScriptCode                      keyScript;
  FileTranslationSpecArrayHandle fileTranslation;
  UInt32                          reserved1;
  CFStringRef                     saveFileName;
  char                            reserved[227];
};
typedef struct NavReplyRecord  NavReplyRecord;
```

Field Descriptions

`validRecord` `true` if the user closed a dialog by clicking the **Open** button in an Open dialog, the **Save** button in a Save Location dialog, or the **Choose** button in a Choose dialog, or by pressing the Return or Enter key. If this field is `false`, the remaining fields do not contain valid data.

`replacing` `true` if the user chooses to save a file by replacing an existing file.

`isStationery` `true` if the file about to be saved should be saved as a stationery document.

`translationNeeded` `true` if translation was or will be needed for selected files in Open and Save Location dialogs.

`selection` An Apple event descriptor list (`AEDescList`) created from `FSSpec` or `FSRef` references to selected items. You can determine the number of items in the list by calling `AECountItems`. Each selected item is described in an `AEDesc` structure by the descriptor type `typeFSS` or `typeFSRef`. This descriptor can be coerced into an `FSSpec` or `FSRef` preparatory to, for example, opening the file.

`keyScript` Keyboard script system used for the filename.

`fileTranslation` Handle to a `FileTranslationSpec` structure, which contains a corresponding translation array for each file reference returned in the `selection` field.

`saveFileName` The save file name in `CFString` form.

This field was introduced with Navigation Services 3.0 because there is no way to fit a 255-character Unicode name into the name field of an FSSpec or into an FSRef. (See selection field.)

Note 1: On Mac OS 9, you will never get a file name that won't fit into the name field of an FSSpec structure.

Note 2: On Mac OS X, you cannot reliably convert the name in the saveFileName field to a 31-byte Pascal string.

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

Responding to User Actions

If the validRecord field of the NavReplyRecord structure contains true, your application typically calls NavDialogGetUserAction to determine the user action, as represented by user action constants. Typical user action constants are as follows:

Constant	Value	Description
kUserActionCancel	1	The user clicked the **Cancel** button.
kUserActionOpen	2	The user clicked the **Open** button in an Open dialog.
kUserActionSaveAs	3	The user clicked the **Save** button in a Save Location dialog.
kUserActionChoose	4	The user clicked the **Choose** button in a Choose dialog

As an alternative to calling NavDialogGetUserAction, you can extract the user action from the userAction field of the NavCBRec structure.

After determining the user action, your event handling function should take the appropriate action. For example, if the **Open** button was clicked, your event handling function should proceed to open the file, or files, selected by the user in the Open dialog.

Note that you should always call the function NavCompleteSave to complete any save operation. Amongst other things, NavCompleteSave performs any needed translation.

kNavCBTerminate Message Received

When the kNavCBTerminate message is received, your event handler should call NavDialogDispose to dispose of the dialog reference.

Event Handling in Primary Alerts

Your event handling function for the primary alerts should be declared in the same way as that for the event handling function for the primary dialogs.

When the kNavCBUserAction message is received, your application should call NavDialogGetUserAction to determine the user action, and then take the appropriate action. The user action constants relevant to the primary alerts are as follows:

Constant	Value	Description
kUserActionCancel	1	The user clicked the **Cancel** button.
kUserActionSaveChanges	6	The user clicked the **Save** button in a Save Changes alert.
kUserActionDontSaveChanges	7	The user clicked the **Don't Save** button in a Save Changes alert.
kUserActionDiscardChanges	8	The user clicked the **OK** button in a Discard Changes alert.

When the appropriate action has been taken, your event handler should call NavDialogDispose to dispose of the dialog reference.

Other Application-Defined (Callback) Functions

Application-Defined 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 (`NavCreateGetFileDialog` or `NavCreateChooseFileDialog`). If you desire more specific filtering, you can provide an application-defined filter (callback) function. Filter functions give you more control over what can and cannot be displayed. You can pass a universal procedure pointer to your filter function in calls to the functions `NavCreateGetFileDialog`, `NavCreateChooseFileDialog`, `NavCreateChooseFolderDialog`, `NavCreateChooseVolumeDialog`, and `NavCreateChooseObjectDialog`.

You can use both a `NavTypeList` structure and a filter function for the Open and Choose a File dialogs 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.

Your filter function should return `true` if an object is to be displayed. The following is an example of a filter function that allows only text files to be displayed:

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

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

  return display;
}
```

Application-Defined (Callback) 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 `inpreviewProc` parameter of the Navigation Services functions `NavCreateGetFileDialog`, `NavCreateChooseFileDialog` and `NavCreateChooseObjectDialog`:

```
Boolean  myPreviewProc(NavCBRecPtr callBackParms,void *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.

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.

Main File Manager Constants, Data Types and Functions

Constants

Read/Write Permission

```
fsCurPerm    = 0
fsRdPerm     = 1
fsWrPerm     = 2
fsRdWrPerm   = 3
fsRdWrShPerm = 4
```

File Mark Positioning Modes

```
fsAtMark     = 0
fsFromStart  = 1
fsFromLEOF   = 2
fsFromMark   = 3
rdVerify     = 64
```

Data Types

File System Specification Structure

```
struct FSSpec
{
  short          vRefNum;     // Volume reference number.
  long           parID;       // Directory ID of parent directory.
  Str63          name;        // Filename or directory name.
};
typedef struct FSSpec FSSpec;
typedef FSSpec *FSSpecPtr, **FSSpecHandle;
```

File System Reference

```
struct FSRef
{
  UInt8 hidden[80];
}
```

File Information Structure

```
struct FInfo
{
  OSType         fdType;      // File type.
  OSType         fdCreator;   // File's creator.
  unsigned short fdFlags;     // Finder flags (fHasBundle, fInvisible, etc).
  Point          fdLocation;  // Position of top-left corner of file's icon.
  short          fdFldr;      // Folder containing file.
};
typedef struct FInfo FInfo;
```

Functions

Reading, Writing and Closing Files

```
OSErr  FSClose(short refNum);
OSErr  FSRead(short refNum,long *count,void *buffPtr);
OSErr  FSWrite(short refNum,long *count,const void *buffPtr);
```

Manipulating the File Mark

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

Manipulating the End-Of-File

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

Opening and Creating Files

```
OSErr  FSpOpenDF(const FSSpec *spec,SInt8 permission,short *refNum);
OSErr  FSpOpenRF(const FSSpec *spec,SInt8 permission,short *refNum);
OSErr  FSpCreate(const FSSpec *spec,OSType creator,OSType fileType,ScriptCode scriptTag);
OSErr  FSCreateFileUnicode(const FSRef *parentRef,UniCharCount nameLength,const UniChar *name,
       FSCatalogInfoBitmap whichInfo,const FSCatalogInfo *catalogInfo,FSRef *newRef,
       FSSpec *newSpec);
```

Deleting Files and Directories

```
OSErr  FSpDelete(const FSSpec *spec);
OSErr  FSDeleteObject(const FSRef *ref);
```

Exchanging Data in Two Files

```
OSErr  FSpExchangeFiles(const FSSpec *source,const FSSpec *dest);
OSErr  FSExchangeObjects(const FSRef *ref,const FSRef *destRef);
```

Creating File System Specifications and File System References

```
OSErr  FSMakeFSSpec(short vRefNum,long dirID,ConstStr255Param fileName,FSSpec *spec);
OSErr  FSpMakeFSRef(const FSSpec *source,FSRef *newRef);
OSErr  FSMakeFSRefUnicode(const FSRef *parentRef,UniCharCount nameLength,const UniChar *name,
       TextEncoding textEncodingHint,FSRef *newRef)
```

Obtaining Volume Information

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

Getting and Setting Finder Information

```
OSErr  FSpGetFInfo(FSSpec *spec,FInfo *fndrInfo);
OSErr  FSpSetFInfo(const FSSpec *spec,const FInfo *fndrInfo);
```

Relevant Resource Manager Functions

Creating and Opening Resource Files

```
void   FSpCreateResFile(const FSSpec *spec,OSType creator,OSType fileType,
       ScriptCode scriptTag);
short  FSpOpenResFile(const FSSpec *spec,SignedByte permission);
```

Relevant Finder Interface Functions

Find a Specified Folder

```
OSErr  FindFolder(short vRefNum,OSType folderType,Boolean createFolder,
       short *foundVRefNum,long *foundDirID)
```

Main Navigation Services Constants, Data Types, and Functions

Constants

Dialog Option Flags

```
kNavDefaultNavDlogOptions  = 0x000000E4
kNavNoTypePopup            = 0x00000001
kNavDontAutoTranslate      = 0x00000002
kNavDontAddTranslateItems  = 0x00000004
kNavAllFilesInPopup        = 0x00000010
kNavAllowStationery        = 0x00000020
kNavAllowPreviews          = 0x00000040
kNavAllowMultipleFiles     = 0x00000080
kNavAllowInvisibleFiles    = 0x00000100
kNavDontResolveAliases     = 0x00000200
kNavSelectDefaultLocation  = 0x00000400
kNavSelectAllReadableItem  = 0x00000800
kNavSupportPackages        = 0x00001000
kNavAllowOpenPackages      = 0x00002000
```

```
kNavDontAddRecents         = 0x00004000
kNavDontUseCustomFrame     = 0x00008000
kNavDontConfirmReplacement = 0x00010000
```

Event Messages

```
kNavCBEvent           = 0
kNavCBCustomize       = 1
kNavCBStart           = 2
kNavCBTerminate       = 3
kNavCBAdjustRect      = 4
kNavCBNewLocation     = 5
kNavCBShowDesktop     = 6
kNavCBSelectEntry     = 7
kNavCBPopupMenuSelect = 8
kNavCBAccept          = 9
kNavCBCancel          = 10
kNavCBAdjustPreview   = 11
kNavCBUserAction      = 12
kNavCBOpenSelection   = (long) 0x80000000
```

User Action

```
kNavUserActionNone            = 0
kNavUserActionCancel          = 1
kNavUserActionOpen            = 2
kNavUserActionSaveAs          = 3
kNavUserActionChoose          = 4
kNavUserActionNewFolder       = 5
kNavUserActionSaveChanges     = 6
kNavUserActionDontSaveChanges = 7
kNavUserActionDiscardChanges  = 8
```

Save Changes Action

```
kNavSaveChangesClosingDocument     = 1
kNavSaveChangesQuittingApplication = 2
```

Data Types

```
typedef struct __NavDialog *NavDialogRef;
typedef UInt32              NavDialogOptionFlags;
typedef SInt32              NavEventCallbackMessage;
typedef void               *NavCallBackUserData;
typedef UInt32              NavUserAction;
typedef UInt32              NavAskSaveChangesAction;
```

NavDialogCreationOptions

```
struct NavDialogCreationOptions
{
  UInt16               version;
  NavDialogOptionFlags optionFlags;
  Point                location;
  CFStringRef          clientName;
  CFStringRef          windowTitle;
  CFStringRef          actionButtonLabel;
  CFStringRef          cancelButtonLabel;
  CFStringRef          saveFileName;
  CFStringRef          message;
  UInt32               preferenceKey;
  CFArrayRef           popupExtension;
  WindowModality       modality;
  WindowRef            parentWindow;
  char                 reserved[16];
};
typedef struct NavDialogCreationOptions NavDialogCreationOptions;
```

NavTypeList

```
struct NavTypeList
{
  OSType componentSignature;
```

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

NavCBRec

```
struct NavCBRec
{
  UInt16        version;
  NavDialogRef  context;
  WindowRef     window;
  Rect          customRect;
  Rect          previewRect;
  NavEventData  eventData;
  NavUserAction userAction;
  char          reserved[218];
};
typedef struct NavCBRec NavCBRec;
typedef NavCBRec *NavCBRecPtr;
```

NavReplyRecord

```
struct NavReplyRecord
{
  UInt16                        version;
  Boolean                       validRecord;
  Boolean                       replacing;
  Boolean                       isStationery;
  Boolean                       translationNeeded;
  AEDescList                    selection;
  ScriptCode                    keyScript;
  FileTranslationSpecArrayHandle fileTranslation;
  UInt32                        reserved1;
  CFStringRef                   saveFileName;
  char                          reserved[227];
};
typedef struct NavReplyRecord  NavReplyRecord;
```

Functions

Initialising the NavDialogCreationOptions Structure

```
OSStatus      NavGetDefaultDialogCreationOptions(NavDialogCreationOptions *outOptions);
```

Creating and Disposing Of Navigation Services Dialogs

```
OSStatus      NavCreateGetFileDialog(const NavDialogCreationOptions *inOptions,
              NavTypeListHandle inTypeList,NavEventUPP inEventProc,NavPreviewUPP inPreviewProc,
              NavObjectFilterUPP inFilterProc,void *inClientData,NavDialogRef *outDialog);
OSStatus      NavCreatePutFileDialog(const NavDialogCreationOptions *inOptions,
              OSType inFileType,OSType inFileCreator,NavEventUPP inEventProc,
              void *inClientData,NavDialogRef *outDialog);
OSStatus      NavCreateAskSaveChangesDialog(const NavDialogCreationOptions *inOptions,
              NavAskSaveChangesAction inAction,NavEventUPP inEventProc,void *inClientData,
              NavDialogRef *outDialog);
OSStatus      NavCreateAskDiscardChangesDialog(const NavDialogCreationOptions *inOptions,
              NavEventUPP inEventProc,void *inClientData,NavDialogRef *outDialog);
OSStatus      NavCreateChooseFileDialog(const NavDialogCreationOptions *inOptions,
              NavTypeListHandle inTypeList,NavEventUPP inEventProc,NavPreviewUPP inPreviewProc,
              NavObjectFilterUPP inFilterProc,void *inClientData,NavDialogRef *outDialog);
OSStatus      NavCreateChooseVolumeDialog(const NavDialogCreationOptions *inOptions,
              NavEventUPP inEventProc,NavObjectFilterUPP inFilterProc,void *inClientData,
              NavDialogRef *outDialog);
OSStatus      NavCreateChooseObjectDialog(const NavDialogCreationOptions *inOptions,
              NavEventUPP inEventProc,NavPreviewUPP inPreviewProc,
              NavObjectFilterUPP inFilterProc,void *inClientData,NavDialogRef *outDialog);
void          NavDialogDispose(NavDialogRef inDialog);
```

Displaying and Running a Navigation Services Dialog

```
OSStatus      NavDialogRun(NavDialogRef inDialog);
```

Filling In and Disposing Of NavReplyRecord Structures

```
OSStatus      NavDialogGetReply(NavDialogRef inDialog,NavReplyRecord *outReply);
OSErr         NavDisposeReply(NavReplyRecord *reply);
```

Getting the User Action

```
NavUserAction NavDialogGetUserAction(NavDialogRef inDialog);
```

Getting and Setting the Save File Name

```
CFStringRef   NavDialogGetSaveFileName(NavDialogRef inPutFileDialog);
OSStatus      NavDialogSetSaveFileName(NavDialogRef inPutFileDialog,CFStringRef inFileName);
```

Completing a Save Operation

```
OSErr         NavCompleteSave(NavReplyRecord *reply,NavTranslationOptions howToTranslate);
```

Getting the Window In Which a Navigation Services Dialog Appears

```
WindowRef     NavDialogGetWindow(NavDialogRef inDialog);
```

Creating New Folders

```
OSStatus      NavCreateNewFolderDialog(const NavDialogCreationOptions *inOptions,
              NavEventUPP inEventProc,void *inClientData,NavDialogRef *outDialog);
```

Creating Previews

```
OSErr         NavCreatePreview(AEDesc *theObject,OSType previewDataType,
              const void *previewData,Size  previewDataSize);
```

Creating and Disposing of Universal Procedure Pointers

```
NavEventUPP         NewNavEventUPP(NavEventProcPtr userRoutine);
NavPreviewUPP       NewNavPreviewUPP(NavPreviewProcPtr userRoutine);
NavObjectFilterUPP NewNavObjectFilterUPP(NavObjectFilterProcPtr userRoutine);
void                DisposeNavEventUPP(NavEventUPP userUPP);
void                DisposeNavPreviewUPP(NavPreviewUPP userUPP);
void                DisposeNavObjectFilterUPP(NavObjectFilterUPP userUPP);
```

Application-Defined (Callback) Functions - Event Handling, Previews, and Filters

```
void                myNavEventFunction(NavEventCallbackMessage callBackSelector,
                    NavCBRecPtr callBackParms,void *callBackUD);
Boolean             myNavPreviewFunction(NavCBRecPtr callBackParms,void *callBackUD);
Boolean             myNavObjectFilterFunction(AEDesc *theItem,void *info,void *callBackUD,
                    NavFilterModes filterMode);
```

Demonstration Program Files Listing

```
// ****************************************************************************************
// Files.h                                                                  CARBON EVENT MODEL
// ****************************************************************************************
//
// This program demonstrates:
//
// •  File operations associated with:
//
//    •  The user invoking the Open…, Close, Save, Save As…, Revert, and Quit commands of a
//      typical application.
//
//    •  Handling of the required Apple events Open Application, Re-open Application, Open
//      Documents, Print Documents, and Quit Application.
//
// •  File synchronisation.
//
// •  The creation, display, and handling of Open, Save Location, Choose a Folder, Save
//    Changes, Discard Changes, and Review Unsaved dialogs and alerts using the new model
//    introduced with Navigation Services 3.0.
//
// To keep the code not specifically related to files and file-handling to a minimum, an item
// is included in the Demonstration menu which allows the user to simulate "touching" a window
// (that is, modifying the contents of the associated document).  Choosing the first menu item
// in this menu sets the window-touched flag in the window's document structure to true and
// draws the text "WINDOW TOUCHED" in the window in a large font size, this latter so that the
// user can keep track of which windows have been "touched".
//
// This program is also, in part, an extension of the demonstration program Windows2 in that
// it also demonstrates certain file-related Window Manager features introduced with the Mac
// OS 8.5 Window Manager.  These features are:
//
// •  Window proxy icons.
//
// •  Window path pop-up menus.
//
// Those sections of the source code relating to these features are identified with ///// at
// the right of each line.
//
// The program utilises the following resources:
//
// •  A 'plst' resource containing an information property list which provides information
//    to the Mac OS X Finder.
//
// •  An 'MBAR' resource, and 'MENU' and 'xmnu' resources for Apple, File, Edit and
//    Demonstration menus (preload, non-purgeable).
//
// •  A 'STR ' resource containing the "missing application name" string, which is copied to
//    all document files created by the program.
//
// •  'STR#' resources (purgeable) containing error strings, the application's name (for
//    certain Navigation Services functions), and a message  string for the Choose a Folder
//    dialog.
//
// •  An 'open' resource (purgeable) containing the file type list for the Open dialog.
//
// •  A 'kind' resource (purgeable) describing file types, which is used by Navigation
//    Services to  build the native file types section of the Show pop-up menu in the Open
//    dialog.
//
// •  Two 'pnot' resources (purgeable) which, together with an associated 'PICT' resource
//    (purgeable) and a 'TEXT' resource created by the program, provide the previews for
//    the PICT and, on Mac OS 8/9, TEXT files.
//
// •  The 'BNDL' resource (non-purgeable), 'FREF' resources (non-purgeable), signature
//    resource (non-purgeable), and icon family resources (purgeable), required to support the
//    built application on Mac OS 8/9.
```

```
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, canBackground,
//    doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// ********************************************************************************************

// ..... includes

#include <Carbon.h>

// ..... defines

#define rMenubar                  128
#define mAppleApplication         128
#define  Apple_About              'abou'
#define mFile                     129
#define  File_New                 'new '
#define  File_Open                'open'
#define  File_Close               'clos'
#define  File_Save                'save'
#define  File_SaveAs              'sava'
#define  File_Revert              'reve'
#define  File_Quit                'quit'
#define  iQuit                    12
#define mDemonstration            131
#define  Demo_TouchWindow         'touc'
#define  Demo_ChooseAFolderDialog 'choo'
#define rErrorStrings             128
#define  eInstallHandler          1000
#define  eMaxWindows              1001
#define  eCantFindFinderProcess   1002                                          /////
#define rMiscStrings              129
#define  sApplicationName         1
#define  sChooseAFolder           2
#define rOpenResource             128
#define kMaxWindows               10
#define kFileCreator              'Kjbb'
#define kFileTypeTEXT             'TEXT'
#define kFileTypePICT             'PICT'
#define kOpen                     0
#define kPrint                    1
#define MIN(a,b)                  ((a) < (b) ? (a) : (b))

// ..... typedefs

typedef struct
{
  TEHandle     editStrucHdl;
  PicHandle    pictureHdl;
  SInt16       fileRefNum;
  FSSpec       fileFSSpec;
  AliasHandle  aliasHdl;
  Boolean      windowTouched;
  NavDialogRef modalToWindowNavDialogRef;
  NavEventUPP  askSaveDiscardEventFunctionUPP;
  Boolean      isAskSaveChangesDialog;
}  docStructure, *docStructurePointer, **docStructureHandle;

// ..... function prototypes

void      main                        (void);
void      eventLoop                   (void);
void      doPreliminaries             (void);
void      doInstallAEHandlers         (void);
OSStatus  appEventHandler             (EventHandlerCallRef,EventRef,void *);
OSStatus  windowEventHandler          (EventHandlerCallRef,EventRef,void *);
void      doIdle                      (void);
void      doDrawContent               (WindowRef);
void      doMenuChoice                (MenuCommand);
```

```
void      doAdjustMenus                (void);
void      doErrorAlert                 (SInt16);
void      doCopyPString                (Str255,Str255);
void      doConcatPStrings             (Str255,Str255);
void      doTouchWindow                (void);
OSErr     openAppEventHandler          (AppleEvent *,AppleEvent *,SInt32);
OSErr     reopenAppEventHandler        (AppleEvent *,AppleEvent *,SInt32);
OSErr     openAndPrintDocsEventHandler (AppleEvent *,AppleEvent *,SInt32);
OSErr     quitAppEventHandler          (AppleEvent *,AppleEvent *,SInt32);
OSErr     doHasGotRequiredParams       (AppleEvent *);
SInt16    doReviewChangesAlert         (SInt16);

OSErr     doNewCommand                 (void);
OSErr     doOpenCommand                (void);
OSErr     doCloseCommand               (NavAskSaveChangesAction);
OSErr     doSaveCommand                (void);
OSErr     doSaveAsCommand              (void);
OSErr     doRevertCommand              (void);

OSErr     doNewDocWindow               (Boolean,OSType,WindowRef *);
EventHandlerUPP  doGetHandlerUPP       (void);
OSErr     doCloseDocWindow             (WindowRef);
OSErr     doOpenFile                   (FSSpec,OSType);
OSErr     doReadTextFile               (WindowRef);
OSErr     doReadPictFile               (WindowRef);
OSErr     doCreateAskSaveChangesDialog (WindowRef,docStructureHandle,NavAskSaveChangesAction);
OSErr     doSaveUsingFSSpec            (WindowRef,NavReplyRecord *);
OSErr     doSaveUsingFSRef             (WindowRef,NavReplyRecord *);
OSErr     doWriteFile                  (WindowRef);
OSErr     doWriteTextData              (WindowRef,SInt16);
OSErr     doWritePictData              (WindowRef,SInt16);

void  getFilePutFileEventFunction   (NavEventCallbackMessage,NavCBRecPtr,NavCallBackUserData);
void  askSaveDiscardEventFunction   (NavEventCallbackMessage,NavCBRecPtr,NavCallBackUserData);

OSErr     doCopyResources              (WindowRef);
OSErr     doCopyAResource              (ResType,SInt16,SInt16,SInt16);

void      doSynchroniseFiles           (void);
OSErr     doChooseAFolderDialog        (void);

// ********************************************************************************************
// Files.c
// ********************************************************************************************

// ..... includes

#include "Files.h"

// ..... global variables

Boolean     gRunningOnX = false;
SInt16      gAppResFileRefNum;
NavEventUPP gGetFilePutFileEventFunctionUPP ;
Boolean     gQuittingApplication = false;

extern SInt16 gCurrentNumberOfWindows;
extern Rect   gDestRect,gViewRect;

// ********************************************************************************** main

void  main(void)
{
  MenuBarHandle menubarHdl;
  SInt32        response;
  MenuRef       menuRef;
  EventTypeSpec applicationEvents[] = { { kEventClassApplication, kEventAppActivated    },
                                        { kEventClassCommand,     kEventProcessCommand  },
                                        { kEventClassMenu,        kEventMenuEnableItems } };
```

```
  EventLoopTimerRef timerRef;

  // ..... do preliminaries

  doPreliminaries();

  // ..... save application's resource file file reference number

  gAppResFileRefNum = CurResFile();

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

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

  Gestalt(gestaltMenuMgrAttr,&response);
  if(response & gestaltMenuMgrAquaLayoutMask)
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
    {
      DeleteMenuItem(menuRef,iQuit);
      DeleteMenuItem(menuRef,iQuit - 1);
    }

    gRunningOnX = true;
  }
  else
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
      SetMenuItemCommandID(menuRef,iQuit,kHICommandQuit);
  }

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

  doInstallAEHandlers();

  // ..... install application event handler

  InstallApplicationEventHandler(NewEventHandlerUPP((EventHandlerProcPtr) appEventHandler),
                                 GetEventTypeCount(applicationEvents),applicationEvents,
                                 0,NULL);

  // ..... install a timer (for file synchronisation)

  InstallEventLoopTimer(GetCurrentEventLoop(),0,TicksToEventTime(15),
                        NewEventLoopTimerUPP((EventLoopTimerProcPtr) doIdle),NULL,
                        &timerRef);

  // ..... get universal procedure pointer to main Navigation Services services event function

  gGetFilePutFileEventFunctionUPP  =
                                NewNavEventUPP((NavEventProcPtr) getFilePutFileEventFunction);

  // ..... run application event loop

  RunApplicationEventLoop();
}

// ********************************************************************** doPreliminaries

void  doPreliminaries(void)
{
  MoreMasterPointers(448);
  InitCursor();
```

```
}

// ********************************************************************** doInstallAEHandlers

void  doInstallAEHandlers(void)
{
  OSErr osError;

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

  osError = AEInstallEventHandler(kCoreEventClass,kAEReopenApplication,
                NewAEEventHandlerUPP((AEEventHandlerProcPtr) reopenAppEventHandler),
                0L,false);
  if(osError != noErr)  doErrorAlert(eInstallHandler);

  osError = AEInstallEventHandler(kCoreEventClass,kAEOpenDocuments,
                NewAEEventHandlerUPP((AEEventHandlerProcPtr) openAndPrintDocsEventHandler),
                kOpen,false);
  if(osError != noErr)  doErrorAlert(eInstallHandler);

  osError = AEInstallEventHandler(kCoreEventClass,kAEPrintDocuments,
                NewAEEventHandlerUPP((AEEventHandlerProcPtr) openAndPrintDocsEventHandler),
                kPrint,false);
  if(osError != noErr)  doErrorAlert(eInstallHandler);

  osError = AEInstallEventHandler(kCoreEventClass,kAEQuitApplication,
                NewAEEventHandlerUPP((AEEventHandlerProcPtr) quitAppEventHandler),
                0L,false);
  if(osError != noErr)  doErrorAlert(eInstallHandler);
}

// ************************************************************************** appEventHandler

OSStatus  appEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                          void * userData)
{
  OSStatus      result = eventNotHandledErr;
  UInt32        eventClass;
  UInt32        eventKind;
  HICommand     hiCommand;
  MenuID        menuID;
  MenuItemIndex menuItem;

  eventClass = GetEventClass(eventRef);
  eventKind  = GetEventKind(eventRef);

  switch(eventClass)
  {
    case kEventClassApplication:
      if(eventKind == kEventAppActivated)
        SetThemeCursor(kThemeArrowCursor);
      break;

    case kEventClassCommand:
      if(eventKind == kEventProcessCommand)
      {
        GetEventParameter(eventRef,kEventParamDirectObject,typeHICommand,NULL,
                          sizeof(HICommand),NULL,&hiCommand);
        menuID = GetMenuID(hiCommand.menu.menuRef);
        menuItem = hiCommand.menu.menuItemIndex;
        if((hiCommand.commandID != kHICommandQuit) &&
           (menuID >= mAppleApplication && menuID <= mDemonstration))
        {
          doMenuChoice(hiCommand.commandID);
          result = noErr;
        }
```

```
      }
      break;

    case kEventClassMenu:
      if(eventKind == kEventMenuEnableItems)
      {
        doAdjustMenus();
        result = noErr;
      }
      break;
  }

  return result;
}

// ********************************************************************** windowEventHandler

OSStatus  windowEventHandler(EventHandlerCallRef eventHandlerCallRef,EventRef eventRef,
                             void* userData)
{
  OSStatus  result = eventNotHandledErr;
  UInt32    eventClass;
  UInt32    eventKind;
  WindowRef windowRef;

  eventClass = GetEventClass(eventRef);
  eventKind  = GetEventKind(eventRef);

  switch(eventClass)
  {
    case kEventClassWindow:
      GetEventParameter(eventRef,kEventParamDirectObject,typeWindowRef,NULL,sizeof(windowRef),
                        NULL,&windowRef);
      switch(eventKind)
      {
        case kEventWindowDrawContent:
          doDrawContent(windowRef);
          result = noErr;
          break;

        case kEventWindowClose:
          if(gQuittingApplication)
            doCloseCommand(kNavSaveChangesQuittingApplication);
          else
            doCloseCommand(kNavSaveChangesClosingDocument);
          result = noErr;
          break;
      }
      break;
  }

  return result;
}

// ********************************************************************************** doIdle

void  doIdle(void)
{
  if(GetWindowKind(FrontWindow()) == kApplicationWindowKind)
    doSynchroniseFiles();
}

// ******************************************************************************** doUpdate

void  doDrawContent(WindowRef windowRef)
{
  docStructureHandle docStrucHdl;
  GrafPtr            oldPort;
  Rect               destRect;
```

```
  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  GetPort(&oldPort);
  SetPortWindowPort(windowRef);

  if((*docStrucHdl)->pictureHdl)
  {
    destRect = (*(*docStrucHdl)->pictureHdl)->picFrame;
    OffsetRect(&destRect,170,54);
    HLock((Handle) (*docStrucHdl)->pictureHdl);
    DrawPicture((*docStrucHdl)->pictureHdl,&destRect);
    HUnlock((Handle) (*docStrucHdl)->pictureHdl);
  }
  else if((*docStrucHdl)->editStrucHdl)
  {
    HLock((Handle) (*docStrucHdl)->editStrucHdl);
    TEUpdate(&gDestRect,(*docStrucHdl)->editStrucHdl);
    HUnlock((Handle) (*docStrucHdl)->editStrucHdl);
  }

  if((*docStrucHdl)->windowTouched)
  {
    TextSize(48);
    MoveTo(30,170);
    DrawString("\pWINDOW TOUCHED");
    TextSize(12);
  }

  SetPort(oldPort);
}

// ********************************************************************************* doMenuChoice

void  doMenuChoice(MenuCommand commandID)
{
  OSErr osError = noErr;

  switch(commandID)
  {
    // ..... Apple/Application menu

    case Apple_About:
      SysBeep(10);
      break;

    // ..... File menu

    case File_New:
      if(osError = doNewCommand())
        doErrorAlert(osError);
      break;

    case File_Open:
      if(osError = doOpenCommand() && osError == opWrErr)
        doErrorAlert(osError);
      break;

    case File_Close:
      if(osError = doCloseCommand(kNavSaveChangesClosingDocument))
        doErrorAlert(osError);
      break;

    case File_Save:
      if(osError = doSaveCommand())
        doErrorAlert(osError);
      break;

    case File_SaveAs:
```

```
      if(osError = doSaveAsCommand())
        doErrorAlert(osError);
      break;

    case File_Revert:
      if(osError = doRevertCommand())
        doErrorAlert(osError);
      break;

    // ..... Demonstration menu

    case Demo_TouchWindow:
      doTouchWindow();
      break;

    case Demo_ChooseAFolderDialog:
      if(osError = doChooseAFolderDialog())
        doErrorAlert(osError);
      break;
  }
}

// ********************************************************************************* doAdjustMenus

void  doAdjustMenus(void)
{
  OSErr              osError;
  MenuRef            menuRef;
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;

  if(gCurrentNumberOfWindows > 0)
  {
    if(gRunningOnX)
    {
      if((osError = GetSheetWindowParent(FrontWindow(),&windowRef)) == noErr)
      {
        menuRef = GetMenuRef(mFile);
        DisableMenuCommand(menuRef,File_Close);
        DisableMenuCommand(menuRef,File_Save);
        DisableMenuCommand(menuRef,File_SaveAs);
        DisableMenuCommand(menuRef,File_Revert);
        menuRef = GetMenuRef(mDemonstration);
        DisableMenuCommand(menuRef,Demo_TouchWindow);
        return;
      }
      else
        windowRef = FrontWindow();
    }
    else
      windowRef = FrontWindow();

    if(GetWindowKind(windowRef) == kApplicationWindowKind)
    {
      docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

      menuRef = GetMenuRef(mFile);
      EnableMenuCommand(menuRef,File_Close);
      if((*docStrucHdl)->windowTouched)
      {
        EnableMenuCommand(menuRef,File_Save);
        EnableMenuCommand(menuRef,File_Revert);
      }
      else
      {
        DisableMenuCommand(menuRef,File_Save);
        DisableMenuCommand(menuRef,File_Revert);
      }
```

```
      if(((*docStrucHdl)->pictureHdl != NULL) ||
         ((*(*docStrucHdl)->editStrucHdl)->teLength > 0))
        EnableMenuCommand(menuRef,File_SaveAs);
      else
        DisableMenuCommand(menuRef,File_SaveAs);

      menuRef = GetMenuRef(mDemonstration);

      if(((*docStrucHdl)->pictureHdl != NULL) ||
         ((*(*docStrucHdl)->editStrucHdl)->teLength > 0))
      {
        if((*docStrucHdl)->windowTouched == false)
          EnableMenuCommand(menuRef,Demo_TouchWindow);
        else
          DisableMenuCommand(menuRef,Demo_TouchWindow);
      }
      else
        DisableMenuCommand(menuRef,Demo_TouchWindow);
    }
  }
  else
  {
    menuRef = GetMenuRef(mFile);
    DisableMenuCommand(menuRef,File_Close);
    DisableMenuCommand(menuRef,File_Save);
    DisableMenuCommand(menuRef,File_SaveAs);
    DisableMenuCommand(menuRef,File_Revert);
    menuRef = GetMenuRef(mDemonstration);
    DisableMenuCommand(menuRef,Demo_TouchWindow);
  }

  DrawMenuBar();
}

// ********************************************************************************* doErrorAlert

void  doErrorAlert(SInt16 errorCode)
{
  Str255 errorString, theString;
  SInt16 itemHit;

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

  if(errorCode != memFullErr)
  {
    StandardAlert(kAlertCautionAlert,errorString,NULL,NULL,&itemHit);
  }
  else
  {
    StandardAlert(kAlertStopAlert,errorString,NULL,NULL,&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)
{
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;

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

  SetPortWindowPort(windowRef);

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

  (*docStrucHdl)->windowTouched = true;

  SetWindowModified(windowRef,true);                                                    /////
}

// ********************************************************************** openAppEventHandler

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

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

  return osError;
}

// ********************************************************************** reopenAppEventHandler

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

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

```
      osError = doNewCommand();

  return osError;
}

// ************************************************************ openAndPrintDocsEventHandler

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

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

  if(osError == noErr)
  {
    osError = doHasGotRequiredParams(appEvent);
    if(osError == noErr)
    {
      osError = AECountItems(&docList,&numberOfItems);
      if(osError == noErr)
      {
        for(index=1;index<=numberOfItems;index++)
        {
          osError = AEGetNthPtr(&docList,index,typeFSS,&keyWord,&returnedType,
                                &fileSpec,sizeof(fileSpec),&actualSize);
          if(osError == noErr)
          {
            osError = FSpGetFInfo(&fileSpec,&fileInfo);
            if(osError == noErr)
            {
              if(osError = doOpenFile(fileSpec,fileInfo.fdType))
                doErrorAlert(osError);

              if(osError == noErr && handlerRefcon == kPrint)
              {
                // Call printing function here
              }
            }
          }
          else
            doErrorAlert(osError);
        }
      }
    }
    else
      doErrorAlert(osError);

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

  return osError;
}

// ********************************************************************** quitAppEventHandler

OSErr  quitAppEventHandler(AppleEvent *appEvent,AppleEvent *reply,SInt32 handlerRefcon)
{
  OSErr              osError;
  WindowRef          windowRef, previousWindowRef;
```

```
docStructureHandle docStrucHdl;
SInt16             touchedWindowsCount = 0;
EventRef           eventRef;
EventTargetRef     eventTargetRef;
SInt16             itemHit;

osError = doHasGotRequiredParams(appEvent);
if(osError == noErr)
{
  if(FrontWindow())
  {
    // ……… if any window has a sheet, bring to front, play system alert sound, and return

    windowRef = GetFrontWindowOfClass(kSheetWindowClass,true);
    if(windowRef)
    {
      SelectWindow(windowRef);
      SysBeep(10);
      return noErr;
    }

    // ……… count touched windows

    windowRef = FrontWindow();
    do
    {
      docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
      if((*docStrucHdl)->windowTouched == true)
        touchedWindowsCount++;
      previousWindowRef = windowRef;
    } while(windowRef = GetNextWindowOfClass(previousWindowRef,kDocumentWindowClass,true));

    // ……… if no touched windows, simply close down

    if(touchedWindowsCount == 0)
      QuitApplicationEventLoop();

    // ……… if touched windows are present, and if running on OS X

    if(gRunningOnX)
    {

      // …… if one touched window, cause Save Changes alert on that window, close all others

      if(touchedWindowsCount == 1)
      {
        gQuittingApplication = true;
        CreateEvent(NULL,kEventClassWindow,kEventWindowClose,0,kEventAttributeNone,
                    &eventRef);
        eventTargetRef = GetWindowEventTarget(FrontWindow());
        SendEventToEventTarget(eventRef,eventTargetRef);
      }

    // …… if more than one touched window, create Review Changes alert, handle button clicks

    else if(touchedWindowsCount > 1)
    {
      itemHit = doReviewChangesAlert(touchedWindowsCount);

        if(itemHit == kAlertStdAlertOKButton)
        {
          gQuittingApplication = true;
          CreateEvent(NULL,kEventClassWindow,kEventWindowClose,0,kEventAttributeNone,
                      &eventRef);
          eventTargetRef = GetWindowEventTarget(FrontWindow());
          SendEventToEventTarget(eventRef,eventTargetRef);
        }
        else if(itemHit == kAlertStdAlertCancelButton)
          gQuittingApplication = false;
```

```
          else if(itemHit == kAlertStdAlertOtherButton)
            QuitApplicationEventLoop();
        }
      }

      // ……… if touched windows are present, and if running on OS 8/9

      else
      {
        gQuittingApplication = true;
        CreateEvent(NULL,kEventClassWindow,kEventWindowClose,0,kEventAttributeNone,
                    &eventRef);
        eventTargetRef = GetWindowEventTarget(FrontWindow());
        SendEventToEventTarget(eventRef,eventTargetRef);
      }
    }
    else
      QuitApplicationEventLoop();
  }

  return osError;
}

// ********************************************************************* doHasGotRequiredParams

OSErr  doHasGotRequiredParams(AppleEvent *appEvent)
{
  DescType returnedType;
  Size     actualSize;
  OSErr    osError;

  osError = AEGetAttributePtr(appEvent,keyMissedKeywordAttr,typeWildCard,&returnedType,
                              NULL,0,&actualSize);
  if(osError == errAEDescNotFound)
    osError = noErr;
  else if(osError == noErr)
    osError = errAEParamMissed;

  return osError;
}

// ********************************************************************** doReviewChangesAlert

SInt16  doReviewChangesAlert(SInt16 touchedWindowsCount)
{
  AlertStdCFStringAlertParamRec paramRec;
  Str255      messageText1 = "\pYou have ";
  Str255      messageText2 = "\p Files documents with unsaved changes. ";
  Str255      messageText3 = "\pDo you want to review these changes before quitting?";
  Str255      countString;
  CFStringRef messageText;
  CFStringRef informativeText =
              CFSTR("If you don't review your documents, all your changes will be lost.");
  DialogRef       dialogRef;
  DialogItemIndex itemHit;

  NumToString(touchedWindowsCount,countString);
  doConcatPStrings(messageText1,countString);
  doConcatPStrings(messageText1,messageText2);
  doConcatPStrings(messageText1,messageText3);
  messageText = CFStringCreateWithPascalString(NULL,messageText1,CFStringGetSystemEncoding());

  GetStandardAlertDefaultParams(&paramRec,kStdCFStringAlertVersionOne);
  paramRec.movable      = true;
  paramRec.defaultText  = CFSTR("Review Changes…");
  paramRec.cancelText    = CFSTR("Cancel");
  paramRec.otherText    = CFSTR("Discard Changes");

  CreateStandardAlert(kAlertStopAlert,messageText,informativeText,&paramRec,&dialogRef);
```

```
  RunStandardAlert(dialogRef,NULL,&itemHit);

  if(messageText != NULL)
    CFRelease(messageText);

  return itemHit;
}

// ********************************************************************************************
// NewOpenCloseSave.c
// ********************************************************************************************

// ..... includes

#include "Files.h"

// ..... global variables

NavDialogRef gModalToApplicationNavDialogRef;
SInt16       gCurrentNumberOfWindows = 0;
Rect         gDestRect, gViewRect;
Boolean      gCloseDocWindow = false;

extern NavEventUPP gGetFilePutFileEventFunctionUPP;
extern SInt16      gAppResFileRefNum;
extern Boolean     gQuittingApplication;
extern Boolean     gRunningOnX;

// ************************************************************************** doNewCommand

OSErr  doNewCommand(void)
{
  WindowRef windowRef;
  OSErr     osError;
  OSType    documentType = kFileTypeTEXT;

  osError = doNewDocWindow(true,documentType,&windowRef);

  if(osError == noErr)
    SetWindowProxyCreatorAndType(windowRef,kFileCreator,documentType,kUserDomain);       /////

  return osError;
}

// ************************************************************************** doOpenCommand

OSErr  doOpenCommand(void)
{
  OSErr                    osError = noErr;
  NavDialogCreationOptions dialogOptions;
  Str255                   applicationName;
  NavTypeListHandle        fileTypeListHdl = NULL;

  // ..... create application-modal Open dialog

  osError = NavGetDefaultDialogCreationOptions(&dialogOptions);
  if(osError == noErr)
  {
    GetIndString(applicationName,rMiscStrings,sApplicationName);
    dialogOptions.clientName = CFStringCreateWithPascalString(NULL,applicationName,
                                                              CFStringGetSystemEncoding());
    dialogOptions.modality = kWindowModalityAppModal;
    fileTypeListHdl = (NavTypeListHandle) GetResource('open',rOpenResource);

    osError = NavCreateGetFileDialog(&dialogOptions,fileTypeListHdl,
                                     gGetFilePutFileEventFunctionUPP,NULL,NULL,NULL,
                                     &gModalToApplicationNavDialogRef);
    if(osError == noErr && gModalToApplicationNavDialogRef != NULL)
    {
```

```
      osError = NavDialogRun(gModalToApplicationNavDialogRef);
      if(osError != noErr)
      {
        NavDialogDispose(gModalToApplicationNavDialogRef);
        gModalToApplicationNavDialogRef = NULL;
      }
    }

    if(dialogOptions.clientName != NULL)
      CFRelease(dialogOptions.clientName);

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

  return osError;
}

// ********************************************************************************* doCloseCommand

OSErr  doCloseCommand(NavAskSaveChangesAction action)
{
  WindowRef          windowRef;
  SInt16             windowKind;
  docStructureHandle docStrucHdl;
  OSErr              osError = noErr;

  windowRef = FrontWindow();
  windowKind = GetWindowKind(windowRef);

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

      // ..... if window has unsaved changes, create Save Changes alert

      if((*docStrucHdl)->windowTouched == true)
      {
        if(IsWindowCollapsed(windowRef))
          CollapseWindow(windowRef,false);

        osError = doCreateAskSaveChangesDialog(windowRef,docStrucHdl,action);
      }

      // ..... otherwise close file and clean up

      else
        osError = doCloseDocWindow(windowRef);
      break;

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

  return osError;
}

// ********************************************************************************* doSaveCommand

OSErr  doSaveCommand(void)
{
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;
  OSErr              osError = noErr;
  Rect               portRect;

  windowRef = FrontWindow();
```

```
  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  // ……… if the document has a file ref number, write the file, otherwise call doSaveAsCommand

  if((*docStrucHdl)->fileRefNum)
  {
    osError = doWriteFile(windowRef);

    SetPortWindowPort(windowRef);
    GetWindowPortBounds(windowRef,&portRect);
    EraseRect(&portRect);
    InvalWindowRect(windowRef,&portRect);
  }
  else
    osError = doSaveAsCommand();

  if(osError == noErr)                                                              /////
    SetWindowModified(windowRef,false);                                             /////

  return osError;
}

// ********************************************************************** doSaveAsCommand

OSErr  doSaveAsCommand(void)
{
  OSErr                    osError = noErr;
  NavDialogCreationOptions dialogOptions;
  WindowRef                windowRef;
  Str255                   windowTitle, applicationName;
  docStructureHandle       docStrucHdl;
  OSType                   fileType;

  // ……… create window-modal Save Location dialog

  osError = NavGetDefaultDialogCreationOptions(&dialogOptions);
  if(osError == noErr)
  {
    dialogOptions.optionFlags |= kNavNoTypePopup;

    windowRef = FrontWindow();

    GetWTitle(windowRef,windowTitle);
    dialogOptions.saveFileName = CFStringCreateWithPascalString(NULL,windowTitle,
                                                    CFStringGetSystemEncoding());
    GetIndString(applicationName,rMiscStrings,sApplicationName);
    dialogOptions.clientName = CFStringCreateWithPascalString(NULL,applicationName,
                                                   CFStringGetSystemEncoding());
    dialogOptions.parentWindow = windowRef;
    dialogOptions.modality = kWindowModalityWindowModal;

    docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
    if((*docStrucHdl)->editStrucHdl != NULL)
      fileType = kFileTypeTEXT;
    else if((*docStrucHdl)->pictureHdl != NULL)
      fileType = kFileTypePICT;

    HLock((Handle) docStrucHdl);

    osError = NavCreatePutFileDialog(&dialogOptions,fileType,kFileCreator,
                                     gGetFilePutFileEventFunctionUPP ,
                                     windowRef,&(*docStrucHdl)->modalToWindowNavDialogRef);
    HUnlock((Handle) docStrucHdl);

    if(osError == noErr && (*docStrucHdl)->modalToWindowNavDialogRef != NULL)
    {
      osError = NavDialogRun((*docStrucHdl)->modalToWindowNavDialogRef);
      if(osError != noErr)
      {
```

```
        NavDialogDispose((*docStrucHdl)->modalToWindowNavDialogRef);
        (*docStrucHdl)->modalToWindowNavDialogRef = NULL;
      }
    }

    if(dialogOptions.saveFileName != NULL)
      CFRelease(dialogOptions.saveFileName);
    if(dialogOptions.clientName != NULL)
      CFRelease(dialogOptions.clientName);
  }

  return osError;
}

// ********************************************************************************* doRevertCommand

OSErr  doRevertCommand(void)
{
  OSErr                    osError = noErr;
  NavDialogCreationOptions dialogOptions;
  WindowRef                windowRef;
  Str255                   windowTitle;
  docStructureHandle       docStrucHdl;

  // ..... create window-modal Discard Changes alert

  osError = NavGetDefaultDialogCreationOptions(&dialogOptions);
  if(osError == noErr)
  {
    windowRef = FrontWindow();

    GetWTitle(windowRef,windowTitle);
    dialogOptions.saveFileName = CFStringCreateWithPascalString(NULL,windowTitle,
                                                        CFStringGetSystemEncoding());
    dialogOptions.parentWindow = windowRef;
    dialogOptions.modality = kWindowModalityWindowModal;

    docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
    if((*docStrucHdl)->askSaveDiscardEventFunctionUPP != NULL)
    {
      DisposeNavEventUPP((*docStrucHdl)->askSaveDiscardEventFunctionUPP);
      (*docStrucHdl)->askSaveDiscardEventFunctionUPP = NULL;
    }
    (*docStrucHdl)->askSaveDiscardEventFunctionUPP =
                           NewNavEventUPP((NavEventProcPtr) askSaveDiscardEventFunction);

    HLock((Handle) docStrucHdl);

    osError = NavCreateAskDiscardChangesDialog(&dialogOptions,
                                               (*docStrucHdl)->askSaveDiscardEventFunctionUPP,
                                               windowRef,
                                               &(*docStrucHdl)->modalToWindowNavDialogRef);
    HUnlock((Handle) docStrucHdl);

    if(osError == noErr && (*docStrucHdl)->modalToWindowNavDialogRef != NULL)
    {
      osError = NavDialogRun((*docStrucHdl)->modalToWindowNavDialogRef);
      if(osError != noErr)
      {
        NavDialogDispose((*docStrucHdl)->modalToWindowNavDialogRef);
        (*docStrucHdl)->modalToWindowNavDialogRef = NULL;
      }
    }

    if(dialogOptions.saveFileName != NULL)
      CFRelease(dialogOptions.saveFileName);
  }

  return osError;
```

```
}

// ********************************************************************************** doNewDocWindow

OSErr  doNewDocWindow(Boolean showWindow,OSType documentType,WindowRef * windowRef)
{
  OSStatus           osError;
  WindowAttributes   attributes = kWindowStandardHandlerAttribute |
                                  kWindowStandardDocumentAttributes;
  Rect               portRect, contentRect = { 0,0,300,500 };
  docStructureHandle docStrucHdl;
  EventTypeSpec      windowEvents[] = { { kEventClassWindow, kEventWindowDrawContent  },
                                        { kEventClassWindow, kEventWindowClose        },
                                        { kEventClassWindow, kEventWindowClickDragRgn },
                                        { kEventClassWindow, kEventWindowPathSelect   } };

  if(gCurrentNumberOfWindows == kMaxWindows)
    return eMaxWindows;

  // ..... create window, change attributes, reposition, install event handler

  osError = CreateNewWindow(kDocumentWindowClass,attributes,&contentRect,windowRef);
  if(osError != noErr)
    return osError;

  SetWTitle(*windowRef,"\puntitled");
  ChangeWindowAttributes(*windowRef,0,kWindowFullZoomAttribute | kWindowResizableAttribute);
  RepositionWindow(*windowRef,NULL,kWindowCascadeOnMainScreen);
  SetPortWindowPort(*windowRef);

  InstallWindowEventHandler(*windowRef,doGetHandlerUPP(),GetEventTypeCount(windowEvents),
                            windowEvents,0,NULL);

  // ..... attach document structure to window

  if(!(docStrucHdl = (docStructureHandle) NewHandle(sizeof(docStructure))))
  {
    DisposeWindow(*windowRef);
    return MemError();
  }

  SetWRefCon(*windowRef,(SInt32) docStrucHdl);

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

  // ..... if text document, create TextEdit structure

  if(documentType == kFileTypeTEXT)
  {
    UseThemeFont(kThemeSmallSystemFont,smSystemScript);

    GetWindowPortBounds(*windowRef,&portRect);
    gDestRect = portRect;
    InsetRect(&gDestRect,6,6);
    gViewRect = gDestRect;

    MoveHHi((Handle) docStrucHdl);
    HLock((Handle) docStrucHdl);

    if(!((*docStrucHdl)->editStrucHdl = TENew(&gDestRect,&gViewRect)))
    {
      DisposeWindow(*windowRef);
```

```
      DisposeHandle((Handle) docStrucHdl);
      return MemError();
    }

    HUnlock((Handle) docStrucHdl);
  }

  // ..... show window and increment open windows count

  if(showWindow)
    ShowWindow(*windowRef);

  gCurrentNumberOfWindows ++;

  return noErr;
}

// ********************************************************************** doGetHandlerUPP

EventHandlerUPP  doGetHandlerUPP(void)
{
  static EventHandlerUPP windowEventHandlerUPP;

  if(windowEventHandlerUPP == NULL)
    windowEventHandlerUPP = NewEventHandlerUPP((EventHandlerProcPtr) windowEventHandler);

  return windowEventHandlerUPP;
}

// ********************************************************************** doCloseDocWindow

OSErr  doCloseDocWindow(WindowRef windowRef)
{
  docStructureHandle docStrucHdl;
  OSErr              osError = noErr;
  EventRef           eventRef;
  EventTargetRef     eventTargetRef;

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  // ..... close file, flush volume, dispose of window and associated memory

  if((*docStrucHdl)->fileRefNum != 0)
  {
    if(!(osError = FSClose((*docStrucHdl)->fileRefNum)))
    {
      osError = FlushVol(NULL,(*docStrucHdl)->fileFSSpec.vRefNum);
      (*docStrucHdl)->fileRefNum = 0;
    }
  }

  if((*docStrucHdl)->editStrucHdl != NULL)
    TEDispose((*docStrucHdl)->editStrucHdl);
  if((*docStrucHdl)->pictureHdl != NULL)
    KillPicture((*docStrucHdl)->pictureHdl);

  DisposeHandle((Handle) docStrucHdl);
  DisposeWindow(windowRef);

  gCurrentNumberOfWindows --;

  // ..... if quitting application

  if(gQuittingApplication)
  {
    if(FrontWindow() == NULL)
      QuitApplicationEventLoop();
    else
    {
```

```
      CreateEvent(NULL,kEventClassWindow,kEventWindowClose,0,kEventAttributeNone,
                  &eventRef);
      eventTargetRef = GetWindowEventTarget(FrontWindow());
      SendEventToEventTarget(eventRef,eventTargetRef);
    }
  }

  return osError;
}

// ********************************************************************************* doOpenFile

OSErr  doOpenFile(FSSpec fileSpec,OSType documentType)
{
  WindowRef          windowRef;
  OSErr              osError = noErr;
  SInt16             fileRefNum;
  docStructureHandle docStrucHdl;

  // ..... create new window

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

  SetWTitle(windowRef,fileSpec.name);

  // ..... open file's data fork

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

  // ..... store file reference number and FSSpec in window's document structure

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

  // ..... read in the file

  if(documentType == kFileTypeTEXT)
  {
    if(osError = doReadTextFile(windowRef))
      return osError;
  }
  else if(documentType == kFileTypePICT)
  {
    if(osError = doReadPictFile(windowRef))
      return osError;
  }

  // ..... set up window's proxy icon, and show window

  SetWindowProxyFSSpec(windowRef,&fileSpec);                                    /////
  GetWindowProxyAlias(windowRef,&((*docStrucHdl)->aliasHdl));                   /////
  SetWindowModified(windowRef,false);                                           /////

  ShowWindow(windowRef);

  return noErr;
}

// ************************************************************ doCreateAskSaveChangesDialog

OSErr  doCreateAskSaveChangesDialog(WindowRef windowRef,docStructureHandle docStrucHdl,
                                    NavAskSaveChangesAction action)
```

```
{
  OSErr                    osError = noErr;
  NavDialogCreationOptions dialogOptions;
  Str255                   windowTitle, applicationName;

  // ……..…… create window-modal Save Changes Changes dialog

  osError = NavGetDefaultDialogCreationOptions(&dialogOptions);
  if(osError == noErr)
  {
    GetWTitle(windowRef,windowTitle);
    dialogOptions.saveFileName = CFStringCreateWithPascalString(NULL,windowTitle,
                                                                CFStringGetSystemEncoding());

    GetIndString(applicationName,rMiscStrings,sApplicationName);
    dialogOptions.clientName = CFStringCreateWithPascalString(NULL,applicationName,
                                                              CFStringGetSystemEncoding());
    dialogOptions.parentWindow = windowRef;
    dialogOptions.modality = kWindowModalityWindowModal;

    if((*docStrucHdl)->askSaveDiscardEventFunctionUPP != NULL)
    {
      DisposeNavEventUPP((*docStrucHdl)->askSaveDiscardEventFunctionUPP);
      (*docStrucHdl)->askSaveDiscardEventFunctionUPP = NULL;
    }
    (*docStrucHdl)->askSaveDiscardEventFunctionUPP =
                                NewNavEventUPP((NavEventProcPtr) askSaveDiscardEventFunction);

    HLock((Handle) docStrucHdl);

    osError = NavCreateAskSaveChangesDialog(&dialogOptions,action,
                                            (*docStrucHdl)->askSaveDiscardEventFunctionUPP,
                                            windowRef,
                                            &(*docStrucHdl)->modalToWindowNavDialogRef);
    HUnlock((Handle) docStrucHdl);

    if(osError == noErr && (*docStrucHdl)->modalToWindowNavDialogRef != NULL)
    {
      (*docStrucHdl)->isAskSaveChangesDialog = true;

      osError = NavDialogRun((*docStrucHdl)->modalToWindowNavDialogRef);
      if(osError != noErr)
      {
        NavDialogDispose((*docStrucHdl)->modalToWindowNavDialogRef);
        (*docStrucHdl)->modalToWindowNavDialogRef = NULL;
        (*docStrucHdl)->isAskSaveChangesDialog = false;
      }

      if(!gRunningOnX)
      {
        if(gCloseDocWindow)
        {
          osError = doCloseDocWindow(windowRef);
          if(osError != noErr)
            doErrorAlert(osError);
          gCloseDocWindow = false;
        }
      }
    }

    if(dialogOptions.saveFileName != NULL)
      CFRelease(dialogOptions.saveFileName);
    if(dialogOptions.clientName != NULL)
      CFRelease(dialogOptions.clientName);
  }

  return osError;
}
```

```
// ********************************************************************** doSaveUsingFSSpec

OSErr  doSaveUsingFSSpec(WindowRef windowRef,NavReplyRecord *navReplyStruc)
{
  OSErr              osError = noErr;
  AEKeyword          theKeyword;
  DescType           actualType;
  FSSpec             fileSpec;
  Size               actualSize;
  docStructureHandle docStrucHdl;
  OSType             fileType;
  CFStringRef        fileName;
  SInt16             fileRefNum;
  Rect               portRect;

  if((*navReplyStruc).validRecord)
  {
    // ..................................................................................................... get FSSpec

    if((osError = AEGetNthPtr(&(*navReplyStruc).selection,1,typeFSS,&theKeyword,
                              &actualType,&fileSpec,sizeof(fileSpec),&actualSize)) == noErr)
    {
      docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

      // ..................................... get file name, convert to Pascal string, assign to name field of FSSpec

      fileName = NavDialogGetSaveFileName((*docStrucHdl)->modalToWindowNavDialogRef);
      if(fileName != NULL)
        osError = CFStringGetPascalString(fileName,&fileSpec.name[0],sizeof(FSSpec),
                                          CFStringGetSystemEncoding());

      // ..................................................................... if not replacing, first create a new file

      if(!((*navReplyStruc).replacing))
      {
        if((*docStrucHdl)->editStrucHdl != NULL)
          fileType = kFileTypeTEXT;
        else if((*docStrucHdl)->pictureHdl != NULL)
          fileType = kFileTypePICT;

        osError = FSpCreate(&fileSpec,kFileCreator,fileType,(*navReplyStruc).keyScript);
        if(osError != noErr)
        {
          NavDisposeReply(&(*navReplyStruc));
          return osError;
        }
      }

      // ..................................................... assign FSSpec to fileFSSpec field of window's document structure

      (*docStrucHdl)->fileFSSpec = fileSpec;

      // ..................................................................... if file currently exists for document, close it

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

      // ..................................................................... open file's data fork and write file

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

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

```
        // … … … … … … … … … … … … … … … … … … … … … proxy icon and file synchronisation stuff

        SetPortWindowPort(windowRef);                                                    /////
        SetWindowProxyFSSpec(windowRef,&fileSpec);                                       /////
        GetWindowProxyAlias(windowRef,&((*docStrucHdl)->aliasHdl));                      /////
        SetWindowModified(windowRef,false);                                              /////

        // … … … … … … … … … … … … … … … … … … … … … … … … … … … …  write file using safe save

        osError = doWriteFile(windowRef);

        NavCompleteSave(&(*navReplyStruc),kNavTranslateInPlace);
      }
    }
  }

  SetPortWindowPort(windowRef);
  GetWindowPortBounds(windowRef,&portRect);
  EraseRect(&portRect);
  InvalWindowRect(windowRef,&portRect);

  return osError;
}

// ************************************************************************** doSaveUsingFSRef

OSErr  doSaveUsingFSRef(WindowRef windowRef,NavReplyRecord *navReplyStruc)
{
  OSErr              osError = noErr;
  AEDesc             aeDesc;
  Size               dataSize;
  FSRef              fsRefParent, fsRefDelete;
  UniCharCount       nameLength;
  UniChar            *nameBuffer;
  FSSpec             fileSpec;
  docStructureHandle docStrucHdl;
  FInfo              fileInfo;
  SInt16             fileRefNum;
  Rect               portRect;

  osError = AECoerceDesc(&(*navReplyStruc).selection,typeFSRef,&aeDesc);
  if(osError == noErr)
  {
    // .................................................................................... get FSRef

    dataSize = AEGetDescDataSize(&aeDesc);
    if(dataSize > 0)
      osError = AEGetDescData(&aeDesc,&fsRefParent,sizeof(FSRef));
    if(osError == noErr)
    {
      // ................................ get file name from saveFileName field of NavReplyRecord

      nameLength = (UniCharCount) CFStringGetLength((*navReplyStruc).saveFileName);
      nameBuffer = (UniChar *) NewPtr(nameLength);
      CFStringGetCharacters((*navReplyStruc).saveFileName,CFRangeMake(0,nameLength),
                            &nameBuffer[0]);
      if(nameBuffer != NULL)
      {
        // ....................................... if replacing, delete the file being replaced

        if((*navReplyStruc).replacing)
        {
          osError = FSMakeFSRefUnicode(&fsRefParent,nameLength,nameBuffer,
                                    kTextEncodingUnicodeDefault,&fsRefDelete);
          {
            if(osError == noErr)
              osError = FSDeleteObject(&fsRefDelete);
            if(osError == fBsyErr)
```

```
          {
            DisposePtr((Ptr) nameBuffer);
            return osError;
          }
        }
      }

      // …… create file with Unicode name (but it can be written with an FSSpec)

      if(osError == noErr)
      {
        osError = FSCreateFileUnicode(&fsRefParent,nameLength,nameBuffer,kFSCatInfoNone,
                                      NULL,NULL,&fileSpec);
        if(osError == noErr)
        {
          docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

          osError = FSpGetFInfo(&fileSpec,&fileInfo);

          if((*docStrucHdl)->editStrucHdl != NULL)
            fileInfo.fdType = kFileTypeTEXT;
          else if((*docStrucHdl)->pictureHdl != NULL)
            fileInfo.fdType = kFileTypePICT;
          fileInfo.fdCreator = kFileCreator;

          if(osError == noErr)
            osError = FSpSetFInfo(&fileSpec,&fileInfo);

          (*docStrucHdl)->fileFSSpec = fileSpec;

          // …… open file's data fork and write file

          if(osError == noErr)
            osError = FSpOpenDF(&fileSpec,fsRdWrPerm,&fileRefNum);

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

            // … … … … … … … … … … … … … … … … … … … … proxy icon and file synchronisation stuff

            SetPortWindowPort(windowRef);                                          /////
            SetWindowProxyFSSpec(windowRef,&fileSpec);                             /////
            GetWindowProxyAlias(windowRef,&((*docStrucHdl)->aliasHdl));            /////
            SetWindowModified(windowRef,false);                                    /////

            // … … … … … … … … … … … … … … … … … … … … … … … … … … …  write file using safe save

            osError = doWriteFile(windowRef);

            NavCompleteSave(&(*navReplyStruc),kNavTranslateInPlace);
          }
        }
      }
    }

    DisposePtr((Ptr) nameBuffer);
  }

  AEDisposeDesc(&aeDesc);
}

SetPortWindowPort(windowRef);
GetWindowPortBounds(windowRef,&portRect);
EraseRect(&portRect);
InvalWindowRect(windowRef,&portRect);

return osError;
```

```
}

// ********************************************************************************** doWriteFile

OSErr  doWriteFile(WindowRef windowRef)
{
  docStructureHandle docStrucHdl;
  FSSpec             fileSpecActual, fileSpecTemp;
  UInt32             currentTime;
  Str255             tempFileName;
  SInt16             tempFileVolNum, tempFileRefNum;
  SInt32             tempFileDirID;
  OSErr              osError = noErr;

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
  fileSpecActual = (*docStrucHdl)->fileFSSpec;

  GetDateTime(&currentTime);
  NumToString((SInt32) currentTime,tempFileName);

  osError = FindFolder(fileSpecActual.vRefNum,kTemporaryFolderType,kCreateFolder,
                       &tempFileVolNum,&tempFileDirID);
  if(osError == noErr)
    osError = FSMakeFSSpec(tempFileVolNum,tempFileDirID,tempFileName,&fileSpecTemp);
  if(osError == noErr || osError == fnfErr)
    osError = FSpCreate(&fileSpecTemp,'trsh','trsh',smSystemScript);
  if(osError == noErr)
    osError = FSpOpenDF(&fileSpecTemp,fsRdWrPerm,&tempFileRefNum);
  if(osError == noErr)
  {
    if((*docStrucHdl)->editStrucHdl)
      osError = doWriteTextData(windowRef,tempFileRefNum);
    else if((*docStrucHdl)->pictureHdl)
      osError = doWritePictData(windowRef,tempFileRefNum);
  }
  if(osError == noErr)
    osError = FSClose(tempFileRefNum);
  if(osError == noErr)
    osError = FSClose((*docStrucHdl)->fileRefNum);
  if(osError == noErr)
    osError = FSpExchangeFiles(&fileSpecTemp,&fileSpecActual);
  if(osError == noErr)
    osError = FSpDelete(&fileSpecTemp);
  if(osError == noErr)
    osError = FSpOpenDF(&fileSpecActual,fsRdWrPerm,&(*docStrucHdl)->fileRefNum);

  if(osError == noErr)
    osError = doCopyResources(windowRef);

  return osError;
}

// ********************************************************************************** doReadTextFile

OSErr  doReadTextFile(WindowRef windowRef)
{
  docStructureHandle docStrucHdl;
  SInt16             fileRefNum;
  TEHandle           textEditHdl;
  SInt32             numberOfBytes;
  Handle             textBuffer;
  OSErr              osError = noErr;

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
  fileRefNum = (*docStrucHdl)->fileRefNum;

  textEditHdl = (*docStrucHdl)->editStrucHdl;
  (*textEditHdl)->txSize = 10;
  (*textEditHdl)->lineHeight = 15;
```

```
  SetFPos(fileRefNum,fsFromStart,0);
  GetEOF(fileRefNum,&numberOfBytes);

  if(numberOfBytes > 32767)
    numberOfBytes = 32767;

  if(!(textBuffer = NewHandle((Size) numberOfBytes)))
    return MemError();

  osError = FSRead(fileRefNum,&numberOfBytes,*textBuffer);
  if(osError == noErr || osError == eofErr)
  {
    HLockHi(textBuffer);
    TESetText(*textBuffer,numberOfBytes,(*docStrucHdl)->editStrucHdl);
    HUnlock(textBuffer);
    DisposeHandle(textBuffer);
  }
  else
    return osError;

  return noErr;
}

// ************************************************************************** doReadPictFile

OSErr  doReadPictFile(WindowRef windowRef)
{
  docStructureHandle docStrucHdl;
  SInt16             fileRefNum;
  SInt32             numberOfBytes;
  OSErr              osError = noErr;

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
  fileRefNum = (*docStrucHdl)->fileRefNum;

  GetEOF(fileRefNum,&numberOfBytes);
  SetFPos(fileRefNum,fsFromStart,512);
  numberOfBytes -= 512;

  if(!((*docStrucHdl)->pictureHdl = (PicHandle) NewHandle(numberOfBytes)))
    return MemError();

  osError = FSRead(fileRefNum,&numberOfBytes,*(*docStrucHdl)->pictureHdl);
  if(osError == noErr || osError == eofErr)
    return(noErr);
  else
    return osError;
}

// ************************************************************************** doWriteTextData

OSErr  doWriteTextData(WindowRef windowRef,SInt16 tempFileRefNum)
{
  docStructureHandle docStrucHdl;
  TEHandle           textEditHdl;
  Handle             editText;
  SInt32             numberOfBytes;
  SInt16             volRefNum;
  OSErr              osError = noErr;

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
  textEditHdl = (*docStrucHdl)->editStrucHdl;
  editText = (*textEditHdl)->hText;
  numberOfBytes = (*textEditHdl)->teLength;

  osError = SetFPos(tempFileRefNum,fsFromStart,0);
  if(osError == noErr)
    osError = FSWrite(tempFileRefNum,&numberOfBytes,*editText);
```

```
  if(osError == noErr)
    osError = SetEOF(tempFileRefNum,numberOfBytes);
  if(osError == noErr)
    osError = GetVRefNum(tempFileRefNum,&volRefNum);
  if(osError == noErr)
    osError = FlushVol(NULL,volRefNum);

  if(osError == noErr)
    (*docStrucHdl)->windowTouched = false;

  return osError;
}

// ************************************************************************** doWritePictData

OSErr  doWritePictData(WindowRef windowRef,SInt16 tempFileRefNum)
{
  docStructureHandle docStrucHdl;
  PicHandle          pictureHdl;
  SInt32             numberOfBytes, dummyData;
  SInt16             volRefNum;
  OSErr              osError = noErr;

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
  pictureHdl = (*docStrucHdl)->pictureHdl;

  numberOfBytes = 512;
  dummyData = 0;

  osError = SetFPos(tempFileRefNum,fsFromStart,0);

  if(osError == noErr)
    osError = FSWrite(tempFileRefNum,&numberOfBytes,&dummyData);

  numberOfBytes = GetHandleSize((Handle) (*docStrucHdl)->pictureHdl);

  if(osError == noErr)
  {
    HLock((Handle) (*docStrucHdl)->pictureHdl);
    osError = FSWrite(tempFileRefNum,&numberOfBytes,*(*docStrucHdl)->pictureHdl);
    HUnlock((Handle) (*docStrucHdl)->pictureHdl);
  }

  if(osError == noErr)
    osError = SetEOF(tempFileRefNum,512 + numberOfBytes);
  if(osError == noErr)
    osError = GetVRefNum(tempFileRefNum,&volRefNum);
  if(osError == noErr)
    osError = FlushVol(NULL,volRefNum);

  if(osError == noErr)
    (*docStrucHdl)->windowTouched = false;

  return osError;
}

// ************************************************************* getFilePutFileEventFunction

void  getFilePutFileEventFunction(NavEventCallbackMessage callBackSelector,
                                  NavCBRecPtr callBackParms,NavCallBackUserData callBackUD)
{
  OSErr              osError = noErr;
  NavReplyRecord     navReplyStruc;
  NavUserAction      navUserAction;
  SInt32             count, index;
  AEKeyword          theKeyword;
  DescType           actualType;
  FSSpec             fileSpec;
  Size               actualSize;
```

```
FInfo              fileInfo;
OSType             documentType;
WindowRef          windowRef;
AEDesc             aeDesc;
AEKeyword          keyWord;
DescType           typeCode;
Rect               theRect;
Str255             theString, numberString;
docStructureHandle docStrucHdl;

switch(callBackSelector)
{
  case kNavCBUserAction:
    osError = NavDialogGetReply(callBackParms->context,&navReplyStruc);
    if(osError == noErr && navReplyStruc.validRecord)
    {
      navUserAction = NavDialogGetUserAction(callBackParms->context);

      switch(navUserAction)
      {
        // ..... click on Open button in Open dialog

        case kNavUserActionOpen:
          if(gModalToApplicationNavDialogRef != NULL)
          {
            osError = AECountItems(&(navReplyStruc.selection),&count);
            if(osError == noErr)
            {
              for(index=1;index<=count;index++)
              {
                osError = AEGetNthPtr(&(navReplyStruc.selection),index,typeFSS,
                                      &theKeyword,&actualType,&fileSpec,sizeof(fileSpec),
                                      &actualSize);
                if((osError = FSpGetFInfo(&fileSpec,&fileInfo)) == noErr)
                {
                  documentType = fileInfo.fdType;
                  osError = doOpenFile(fileSpec,documentType);
                  if(osError != noErr)
                    doErrorAlert(osError);
                }
              }
            }
          }
          break;

        // ..... click on Save button in Save Location dialog

        case kNavUserActionSaveAs:
          windowRef = callBackUD;
          osError = AECoerceDesc(&navReplyStruc.selection,typeFSRef,&aeDesc);
          if(osError == noErr)
          {
            osError = doSaveUsingFSRef(windowRef,&navReplyStruc);
            if(osError != noErr)
              doErrorAlert(osError);
            AEDisposeDesc(&aeDesc);
          }
          else
          {
            osError = doSaveUsingFSSpec(windowRef,&navReplyStruc);
            if(osError != noErr)
              doErrorAlert(osError);
          }
          break;

        // ..... click on Choose button in Choose a Folder dialog

        case kNavUserActionChoose:
          if((osError = AEGetNthPtr(&(navReplyStruc.selection),1,typeFSS,&keyWord,&typeCode,
```

```
                                    &fileSpec,sizeof(FSSpec),&actualSize)) == noErr)
            {
              FSMakeFSSpec(fileSpec.vRefNum,fileSpec.parID,fileSpec.name,&fileSpec);
            }
            windowRef = callBackUD;
            SetPortWindowPort(windowRef);
            TextSize(10);
            SetRect(&theRect,0,271,600,300);
            EraseRect(&theRect);
            doCopyPString(fileSpec.name,theString);
            doConcatPStrings(theString, "\p   Volume Reference Number: ");
            NumToString((SInt32) fileSpec.vRefNum,numberString);
            doConcatPStrings(theString,numberString);
            doConcatPStrings(theString, "\p   Parent Directory ID: ");
            NumToString((SInt32) fileSpec.parID,numberString);
            doConcatPStrings(theString,numberString);
            MoveTo(10,290);
            DrawString(theString);
            break;
        }

        osError = NavDisposeReply(&navReplyStruc);
      }
      break;

    case kNavCBTerminate:
      if(gModalToApplicationNavDialogRef != NULL)
      {
        NavDialogDispose(gModalToApplicationNavDialogRef);
        gModalToApplicationNavDialogRef = NULL;
      }
      else
      {
        windowRef = callBackUD;
        docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);
        if((*docStrucHdl)->modalToWindowNavDialogRef != NULL)
        {
          NavDialogDispose((*docStrucHdl)->modalToWindowNavDialogRef);
          (*docStrucHdl)->modalToWindowNavDialogRef = NULL;
        }
      }
      break;
  }
}

// ************************************************************ askSaveDiscardEventFunction

void askSaveDiscardEventFunction(NavEventCallbackMessage callBackSelector,
                                 NavCBRecPtr callBackParms,NavCallBackUserData callBackUD)
{
  WindowRef          windowRef;
  docStructureHandle docStrucHdl;
  NavUserAction      navUserAction;
  OSErr              osError = noErr;
  Rect               portRect;

  switch(callBackSelector)
  {
    case kNavCBUserAction:
      windowRef = callBackUD;
      docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

      if((*docStrucHdl)->modalToWindowNavDialogRef != NULL)
      {
        navUserAction = NavDialogGetUserAction(callBackParms->context);
        switch(navUserAction)
        {
          // ..... click on Save button in Save Changes alert
```

```
          case kNavUserActionSaveChanges:
            osError = doSaveCommand();
            if(osError != noErr)
              doErrorAlert(osError);

          // ..... click on Don't Save button in Save Changes alert

          case kNavUserActionDontSaveChanges:
            NavDialogDispose((*docStrucHdl)->modalToWindowNavDialogRef);
            if(gRunningOnX)
            {
              osError = doCloseDocWindow(windowRef);
              if(osError != noErr)
                doErrorAlert(osError);
            }
            else
              gCloseDocWindow = true;
            break;

          // ..... click on OK button in Discard Changes alert

          case kNavUserActionDiscardChanges:
            GetWindowPortBounds(windowRef,&portRect);
            SetPortWindowPort(windowRef);
            EraseRect(&portRect);

            if((*docStrucHdl)->editStrucHdl != NULL && (*docStrucHdl)->fileRefNum != 0)
            {
              osError = doReadTextFile(windowRef);
              if(osError != noErr)
                doErrorAlert(osError);
            }
            else if((*docStrucHdl)->pictureHdl != NULL)
            {
              KillPicture((*docStrucHdl)->pictureHdl);
              (*docStrucHdl)->pictureHdl = NULL;

              osError = doReadPictFile(windowRef);
              if(osError != noErr)
                doErrorAlert(osError);
            }

            (*docStrucHdl)->windowTouched = false;
            SetWindowModified(windowRef,false);                                    /////
            InvalWindowRect(windowRef,&portRect);

            NavDialogDispose((*docStrucHdl)->modalToWindowNavDialogRef);
            (*docStrucHdl)->modalToWindowNavDialogRef = NULL;
            break;

          // ..... click on Cancel button in Save Changes or Discard Changes alert

          case kNavUserActionCancel:
            if((*docStrucHdl)->isAskSaveChangesDialog == true)
            {
              gQuittingApplication = false;
              (*docStrucHdl)->isAskSaveChangesDialog = false;
            }
            NavDialogDispose((*docStrucHdl)->modalToWindowNavDialogRef);
            (*docStrucHdl)->modalToWindowNavDialogRef = NULL;
            break;
        }
      }
      break;
  }
}

// ********************************************************************** doCopyResources
```

```
OSErr  doCopyResources(WindowRef windowRef)
{
  docStructureHandle docStrucHdl;
  OSType             fileType;
  OSErr              osError = noErr;
  SInt16             fileRefNum;
  Handle             editTextHdl, textResourceHdl;

  docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

  if((*docStrucHdl)->editStrucHdl)
    fileType = kFileTypeTEXT;
  else if((*docStrucHdl)->pictureHdl)
    fileType = kFileTypePICT;

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

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

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

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

    if(!gRunningOnX && fileType == kFileTypeTEXT)
    {
      doCopyAResource('pnot',129,gAppResFileRefNum,fileRefNum);

      editTextHdl = (*(*docStrucHdl)->editStrucHdl)->hText;
      textResourceHdl = NewHandleClear(1024);
      BlockMoveData(*editTextHdl,*textResourceHdl,1024);
      UseResFile(fileRefNum);
      AddResource(textResourceHdl,'TEXT',129,"\p");
      if(ResError() == noErr)
        UpdateResFile(fileRefNum);
      ReleaseResource(textResourceHdl);
    }
  }
  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();
}

// ********************************************************************************************
// SynchroniseFiles.c
// ********************************************************************************************

// ..... includes

#include "Files.h"

// ..... global variables

extern SInt16  gCurrentNumberOfWindows;

// ************************************************************************ doSynchroniseFiles

void  doSynchroniseFiles(void)
{
  WindowRef          windowRef;
  SInt16             trashVRefNum;
  SInt32             trashDirID;
  docStructureHandle docStrucHdl;
  Boolean            aliasChanged;
  AliasHandle        aliasHdl;
  FSSpec             newFSSpec;
  OSErr              osError;

  windowRef = FrontNonFloatingWindow();

  while(windowRef != NULL)
  {
    docStrucHdl = (docStructureHandle) GetWRefCon(windowRef);

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

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

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

      osError = FindFolder(kUserDomain,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(windowRef);
            gCurrentNumberOfWindows --;
            break;
          }
        } while(FSMakeFSSpec(newFSSpec.vRefNum,newFSSpec.parID,"\p",&newFSSpec) == noErr);
      }
    }

    windowRef = GetNextWindow(windowRef);
  }
}

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

// ..... includes

#include "Files.h"

// ..... global variables

extern NavEventUPP  gGetFilePutFileEventFunctionUPP ;
extern NavDialogRef gModalToApplicationNavDialogRef;

// ********************************************************************** doChooseAFolderDialog

OSErr  doChooseAFolderDialog(void)
{
  OSErr                    osError = noErr;
  NavDialogCreationOptions dialogOptions;
  WindowRef                windowRef, parentWindowRef;
  Str255                   message;

  osError = NavGetDefaultDialogCreationOptions(&dialogOptions);
  if(osError == noErr)
  {
    if((osError = GetSheetWindowParent(FrontWindow(),&parentWindowRef)) == noErr)
      windowRef = parentWindowRef;
    else
      windowRef = FrontWindow();

    GetIndString(message,rMiscStrings,sChooseAFolder);
    dialogOptions.message = CFStringCreateWithPascalString(NULL,message,
                                                           CFStringGetSystemEncoding());
    dialogOptions.modality = kWindowModalityAppModal;

    osError = NavCreateChooseFolderDialog(&dialogOptions,gGetFilePutFileEventFunctionUPP ,
                                          NULL,windowRef,&gModalToApplicationNavDialogRef);

    if(osError == noErr && gModalToApplicationNavDialogRef != NULL)
    {
      osError = NavDialogRun(gModalToApplicationNavDialogRef);
      if(osError != noErr)
      {
        NavDialogDispose(gModalToApplicationNavDialogRef);
        gModalToApplicationNavDialogRef = NULL;
      }
    }
```

```
  }

  return osError;
}

// ********************************************************************************************
```

Demonstration Program Files Comments

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 that 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 one "touched" window is open, choosing Quit when two or more "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 the PICT Document or the TEXT Document open:

    - With the document's Finder icon 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 the PICT Document and the TEXT Document open and "touched", and several other windows open, choose Restart or Shut Down from the Mac OS 8/9 Finder's Special menu or the Mac OS X Apple menu (thus invoking a Quit Application event), noting that, for "touched" windows, the Save Changes alert is presented asking the user whether the file should be saved before the shutdown process proceeds. (On Mac OS X, a Review Unsaved alert will be presented at first.)

Note, however, that no printing functions are included. Thus, selecting one or more document icons and choosing Print from the Finder's File menu (Mac OS 8/9) will result in the file/s opening but not printing.

Files.h

defines

Constants are established for a 'STR#' resource containing error strings for three specific error conditions, a 'STR#' resource containing the application's name and the message string for the Choose a Folder dialog, and the 'open' resource containing the file types list.

KFileCreator represents the application's signature and the next two constants represent the file types that are readable and writeable by the application.

typedefs

Each window created by the program will have an associated document structure. The docStructure data type will be used for document structures.

The editStrucHdl field will be assigned a handle to a TextEdit 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. 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. The windowTouched field will be set to true when a window has been made "touched".

When modal-to-the-window Navigation Services dialogs (Save Location, Save Changes, and Discard Changes alerts) are created, the dialog reference will be assigned to the modalToWindowNavDialogRef field. When Save Changes and Discard Changes alerts are created, a universal procedure pointer to the associated event (callback) function will be assigned to the askSaveDiscardChangesDialog field. When a Save Changes alert is created, the isAskSaveChangesDialog field will be set to true to enable the associated event (callback) function to re-set a "quitting application" flag if the user clicks the Cancel button in a Save Changes alert (but not if the user clicks the Cancel button in a Discard Changes alert).

Files.c

Global Variables

gAppResFileRefNum will be assigned the file reference number of the application's resource fork. gGetFilePutFileEventFunctionUPP will be assigned a universal procedure pointer to the event (callback) function associated with the Open, Save Location, and Choose a Folder dialogs. gQuittingApplication is set to true in certain circumstances within quitAppEventHandler and to false if the Cancel button is clicked in a Save Changes or Review Unsaved alert.

main

The file reference number of the application's resource fork (which is opened automatically at application launch) is assigned to the global variable gAppResFileRefNum.

After the required Apple event handlers are installed, the program's application event handler and an timer are installed. The timer is set to fire at an interval of 15 ticks, and will be used to trigger calls to the function doIdle, which calls the program's file synchronisation function.

A universal procedure pointer to the event (callback) function associated with the Open, Save Location, and Choose a Folder dialogs is created and assigned to the global variable gGetFilePutFileEventFunctionUPP.

doInstallAEHandlers

doInstallAEHandlers installs handlers for the Open Application, Re-Open Application, Open Document, Print Documents, and Quit Application events. Since the program installs its own Quit Application event handler, the default Quit Application event handler will not be installed when RunApplicationEventLoop is called.

windowEventHandler

windowEventHandler is the program's window event handler (a callback function), which is installed on all document windows.

Note that, when the event type kEventWindowClose is received, the constant passed in the call to doCloseCommand depends on whether the global variable gQuittingApplication is set to true or false. Amongst other things, this constant affects the text in the Save Changes alert.

Note also that no code is required in a Carbon application to handle window path pop-up menus. The standard window handler handles all user interaction with window path pop-up menus, including bringing the Finder to the front when the user chooses a folder.

doIdle

doIdle is called when the installed timer fires. If the front window is a document window, doSynchroniseFiles is called to synchronises the application with the actual current location (and name) of its currently open document files.

doDrawContent

doDrawContent is called when the kEventWindowDrawContent event type is received. It performs such window updating as is necessary for the satisfactory execution of the demonstration aspects of the program.

doMenuChoice

At the File_Close case, kNavSaveChangesClosingDocument is passed in the call to doCloseCommand. This affects the wording in the Save Changes alert.

doAdjustMenus

If the program is running on Mac OS X, GetSheetWindowParent is called as a way of determining whether the frontmost window is a sheet. If it is, the File and Demonstration menus are adjusted accordingly.

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.

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.

openAppEventHandler, reopenAppEventHandler, and openAndPrintDocsEventHandler

The handlers for the first four required Apple events are essentially identical to those in the demonstration program AppleEvents. One major difference is that one handler (openAndPrintDocsEventHandler) is used for both the Open Documents and Print Documents events, with a value passed in the handler's handlerRefcon parameter advising the handler which of the two events has been received.

Most programs should simply open a new untitled window on receipt of an Open Application event. Accordingly, openAppEventHandler 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 openAndPrintDocsEventHandler, 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.)

quitAppEventHandler

Much of the code in quitAppEventHandler has to do with the requirement, on Mac OS X only, to present a Review Unsaved alert if more than one window with unsaved changes is open when the event is received.

If no windows are open, QuitApplicationEventLoop is called to close the application down. If at least one window is open, the following occurs.

GetFrontWindowOfClass is called to determine whether any window has a sheet. If so, that window is brought to the front and activated and the handler returns immediately, keeping the application alive.

The do-while loop walks the window list counting the number of document windows with unsaved changes (that is, "touched" windows) and, at the same time, bringing those windows to the front. At the next block, if there are no touched document windows, QuitApplicationEventLoop is called to close the application down.

If the application is running on Mac OS X, the following occurs:

- If there is only one touched window open, the flag gQuittingApplication is set to true and a kEventWindowClose event is created and sent to the front window. As will be see, this results in a sequence involving doCloseCommand and doCloseDocWindow whereby all untouched windows in front of the touched window are disposed of and a Save Changes alert is presented for the touched window. In this sequence, if the event handler for the Save Changes alert detects a Cancel button click, gQuittingApplication will be set to false, an action which will cause the process of closing down the remaining windows and the application to be terminated. If the Save or Don't Save buttons are clicked, all remaining windows will be closed down, and the program will be closed down by a call to QuitApplicationEventLoop, within the function doCloseDocWindow.

- If more than one window has been touched, doReviewChangesAlert is called to create, display and handle a Review Changes alert. If the Review Changes… button is hit, the flag gQuittingApplication is set to true and a kEventWindowClose event is created and sent to the front window, resulting in the same

general process of close-down, and possible termination of that close-down process, described above. If the Cancel button is hit, the flag gQuittingApplication is set to false (which defeats the execution of the last block of code in doCloseDocWindow) and quitAppEventHandler simply returns. If the Discard Changes button is hit, QuitApplicationEventLoop is called to terminate the program.

If the application is running on Mac OS 8/9, a Review Unsaved alert is not invoked. Instead, a kEventWindowClose event is created and sent to the front window. This results in the the same general process of close-down, and possible termination of that close-down process, described above. If the Cancel button is not clicked in all Save Changes alerts, all windows will be closed down, and QuitApplicationEventLoop called, within the function doCloseDocWindow.

NewOpenCloseSave.c

Global Variables

gModalToApplicationNavDialogRef will be assigned the dialog reference for the Open File dialog, which is made application-modal. gCurrentNumberOfWindows keeps a count of the number of windows opened. gDestRect and gViewRect are used to set the destination and view rectangles for the TextEdit structures associated with 'TEXT' files.

doNewCommand

doNewCommand 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 doNewDocWindow returns 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, which is called when the user chooses Open from the File menu, uses Navigation Services 3.0 functions to create and display a application-modal Open dialog.

NavGetDefaultDialogCreationOptions initialises the specified NavDialogCreationOptions structure with the defaults.

GetIndString retrieves the application's name and assigns it to an Str255 variable. This is then converted to a CFString and assigned to the clientName field of the NavDialogCreationOptions structure. This will cause the application's name to appear in the dialog's title bar.

The next line assigns a value to the modality field of the NavDialogCreationOptions structure which will cause the dialog to be application-modal.

An 'open' resource containing the file type list is then read in and the handle assigned a variable of type NavTypeListHandle. (The 'open' resource specifies that 'TEXT' and 'PICT' file types are supported.)

The call to NavCreateGetFileDialog creates the dialog. Since the default options are being used, multiple file selection is allowed. A universal procedure pointer to the event function getFilePutFileEventFunction, which will respond to button clicks in the dialog, is passed in the third parameter. No preview function or filter function is used, and no user data is passed in. The last parameter (a global variable) receives the dialog reference.

The call to NavDialogRun displays the dialog.

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 Apple event is received.

The first two lines get a reference to the front window and establish whether the front window is a document window or a modeless dialog.

If the front window is a document window, the handle to the window's document structure is retrieved from the window's window object, allowing a check of whether the window is touched (that is, has unsaved changes). If it does, doCreateAskSaveChangesDialog is called to create and display a Save Changes alert and the function returns, otherwise doCloseDocWindow is called. Prior to the call to doCreateAskSaveChangesDialog, if the window is collapsed (Mac OS 8/9) or minimized in the dock (Mac OS X) it is first uncollapsed or brought out of the Dock.

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

doSaveCommand

doSaveCommand is called when the user chooses Save from the File menu or clicks the Save button in a Save Changes alert.

The first two lines get the WindowRef 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 function doWriteFile is called. The next four lines are incidental to the demonstration; they simply remove the words "WINDOW TOUCHED" from the window.

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 uses Navigation Services 3.0 functions to create and display a window-modal Save Location dialog. It 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.

NavGetDefaultDialogCreationOptions initialises the specified NavDialogCreationOptions structure with the defaults. The first line in the if block unsets the "allow saving of stationery files" bit (one of the defaults). On Mac OS 8/9, this means that the dialog will not contain the Format: pop-up menu.

GetWTitle gets the front window's title into an Str255 variable. This is then converted to a CFString and assigned to the saveFileName field of the NavDialogCreationOptions structure. This will be the default name for the saved file and will appear in the Name (OS 8/9) and Save As (OS X) edit text fields in the Save Location dialog.

The next two lines assign the application's name to the clientName field of the NavDialogCreationOptions structure. This will then appear in the dialog's title bar.

The next two lines assign the window reference to the parentWindow field of the NavDialogCreationOptions structure and assign a value to the modality field which will cause the dialog to be window-modal.

The next block gets the file type from the window's document structure into a local variable.

The call to NavCreatePutFileDialog creates the dialog. A universal procedure pointer to the event function getFilePutFileEventFunction, which will respond to button clicks in the dialog, is passed in the fourth parameter. The window reference is passed in the fifth (user data) parameter. This will be passed to the event function. The dialog reference is assigned to a field of the window's document structure.

The call to NavDialogRun displays the dialog.

doRevertCommand

doRevertCommand, which is called when the user chooses Revert from the File menu, uses Navigation Services 3.0 functions to create and display a window-modal Discard Changes alert. The general approach is similar to that used to create and display the Save Location dialog, the main difference being that a universal procedure pointer to the event function askSaveDiscardEventFunction is stored in the askSaveDiscardEventFunctionUPP field of the window's document structure.

doNewDocWindow

doNewDocWindow is called by doNewCommand and doOpenFile.

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

The call to CreateNewWindow creates a new window with the standard document window attributes and with the standard window event handler installed. SetWTitle is called to set the window's title to "untitled".

ChangeWindowAttributes is called to remove the zoom box/button and grow box from the window. The call to InstallWindowEventHandler installs the program's window event handler on the window.

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 block initialises fields of the document structure.

If the document type is 'TEXT', the if block executes, creating a TextEdit structure and assigning a handle to that structure to the editStrucHdl field of the document structure. (Note that the processes here are not explained in detail because TextEdit and TextEdit 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 structure. (TextEdit is addressed in detail at Chapter 21.))

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.

doCloseDocWindow

doCloseDocWindow is called from doCloseCommand when the subject window is not touched and from the Save Changes alert event handler askSaveDiscardEventFunction when the user clicks the Save or Don't Save buttons in a Save Changes alert.

The FSClose call closes the file, and FlushVol stores to disk all unwritten data currently in the volume buffer.

If the document is a text document, the TextEdit structure is disposed of. If it is a picture document, the Picture structure is disposed of. Finally, the document structure and window are disposed of and the global variable which keeps track of the number of open windows is decremented.

The last block executes only if gQuittingApplication has been set to true in the function quitAppEventHandler. If all windows have been closed, QuitApplicationEventLoop is called to terminate the program; otherwise a kEventWindowClose is created and sent to the front window, causing doCloseCommand to be called from the window event handler. This repetitive calling of doCloseCommand and doCloseDocWindow will continue until no windows remain or until gQuittingApplication is set to false by a click in the Cancel button in a Save Changes or, on Mac OS X only, a Review Unsaved alert.

doOpenFile

doOpenFile opens a new document window and calls the functions which read in the file. It is called by the event function getFilePutFileEventFunction when an Open button click occurs in an Open dialog. The event function passes the file system specification structure and document type to doOpenFile.

The call to doNewDocWindow opens a new window and creates an associated document structure. SetWTitle sets the window's title using information in the file system specification structure. 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, 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.

doCreateAskSaveChangesDialog

doCreateAskSaveChangesDialog, which is called from doCloseCommand, uses Navigation Services 3.0 functions to create and display a window-modal Save Changes alert. The general approach is similar to that used to create and display the Discard Changes alert, but note that in this case that the isAskSaveChangesDialog field of the window's document structure is set to true.

Note also that, if the program is running on Mac OS 8/9, and if gCloseDocWindow is true, doCloseDocWindow is called to close the file, flush the volume, and close down the window. (gCloseDocWindow is set to true

in the callback function askSaveDiscardEventFunction if the user clicks the Don't Save push button button in the Save Changes alert.)

doSaveUsingFSSPec

As will be seen in the event function getFilePutFileEventFunction, when the user clicks on the Save button in a Save Location dialog, AECoerceDesc is called on the descriptor structure in the selection field of the NavReplyRecord structure in an attempt to coerce it to type FSRef. If the call is successful (meaning that the program is running on Mac OS X), doSaveUsingFSRef is called to perform the save using the HFS Plus API. If the call is not successful (meaning that the program is running on Mac OS 8/9) this function (doSaveUsingFSSpec) is called.

A descriptor structure is returned in the selection field of the NavReplyRecord structure. AEGetNthPtr coerces the descriptor structure to typeFSS and stores the result in the local variable fileSpec.

The name field of fileSpec will be empty at this stage. Accordingly, the Navigation Services 3.0 function NavDialogGetSaveFileName is called to get a CFStringRef to the filename from the dialog object, which is converted to a Pascal string and assigned to the name field of fileSpec.

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 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 function doWriteFile is called to write the document to the new file. NavCompleteSave is called to complete the save operation.

Just before the call to doWriteFile, 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.

doSaveUsingFSRef

doSaveUsingFSRef, which is called from the event function getFilePutFileEventFunction, performs the save using the HFS Plus API. The main if block executes only if the call to AECoerceDesc is successful in coercing the descriptor structure in the selection field of the NavReplyRecord to type FSRef.

In Carbon, the dataHandle field of descriptor structures is opaque. Thus AEGetDescData is used to extract the data in this field, which is assigned to the local variable fsRefParent. This is the FSRef for the parent directory.

At the next block, CFStringGetLength is called to get the number of 16-bit Unicode characters in the saveFileName field of the NavReplyRecord structure. This facilitates the call to CFStringGetCharacters, which extracts the contents of the string into a buffer.

If the value in the replacing field of the NavReplyRecord structure indicates that the file is being replaced, the existing file is first deleted. FSMakeFSRefUnicode, given a parent directory and Unicode file name, creates an FSRef for the file. This is passed in the call to FSDeleteObject, which deletes the file.

The call to FSCreateFileUnicode creates a new file with the Unicode name. On return, the last parameter contains a file system specification structure for the new file. (Although the file is created with a Unicode name, it can be written using a file system specification structure.)

The call to FSpGetFInfo gets the Finder information from the volume catalog entry for the file. The file type extracted from the window's document structure is then assigned to the fdType field of the returned FInfo structure, following which FSpSetFInfo is called to set the new Finder information in the file's volume catalog entry.

The file system specification structure is assigned to the fileFSSpec field of the window's document structure.

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 function doWriteFile is called to write the document to the new file. NavCompleteSave is called to complete the save operation.

Just before the call to doWriteFile, 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.

doWriteFile

doWriteFile is called by doSaveCommand, doSaveUsingFSSPec, and doSaveUsingFSRef. In conjunction with two supporting functions, it writes the document to disk using the "safe-save" procedure.

The first two lines retrieve a handle to the document structure and the file system specification from the document structure.

The next two lines create a temporary file name which is bound to be unique. FindFolder finds the temporary folder on the file's volume, or creates a temporary folder if necessary. FSMakeFSSpec makes a file system specification structure for the temporary file, using the volume reference number and parent directory ID returned by the FindFolder call. FSpCreate creates the temporary file in that directory on that volume, and FSpOpenDF opens the file's data fork.

Within the next if block, the appropriate function is called to write the document's data to the temporary file.

The two calls to FSClose close both the temporary and existing files prior to the call to FSpExchangeFiles, which swaps the files' data. The temporary file is then deleted and the data fork of the existing file is re-opened.

The 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 the event function askSaveDiscardEventFunction 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 structure from the document structure and modify the text size and line height fields of the TextEdit 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 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 TESetText. TESetText copies the text in the buffer into the existing hText handle of the TextEdit edit structure. The buffer is then unlocked and disposed of.

doReadPictFile

doReadPictFile is called by doOpenFile and the event function askSaveDiscardEventFunction 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 structure from the document structure. The number of bytes of text is then retrieved from the teLength field of the TextEdit 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.

getFilePutFileEventFunction

getFilePutFileEventFunction is the event (callback) function pertaining to the Open, Save Location, and Choose a Folder dialogs. It responds to button clicks in those dialogs.

When the user has clicked one of the dialog's buttons, the kNavCBUserAction message is received in the callBackSelector formal parameter. When this message is received, the first action is to call NavDialogGetReply to get a NavReplyRecord structure containing information about the dialog session. NavDialogGetUserAction is then called to get the user action which dismissed the dialog.

If the user clicked the Open button in an Open dialog, AECountItems is called 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. The for loop repeats for each of the descriptor structures. AEGetNthPtr 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 function doOpenFile, which creates a new window and reads in the file.

If the user clicked the Save button in a Save Location dialog, the window reference received in the callBackUD formal parameter is assigned to the local variable windowRef. (Recall that the window reference for the front window was passed in the fifth parameter of the call to NavCreatePutFileDialog.) The next task is to determine which of the two file saving functions (doSaveUsingFSSpec or doSaveUsingFSRef) should be called to save the file. Accordingly, AECoerceDesc is called in an attempt to coerce the descriptor structure in the selection field of the NavReplyRecord structure to type FSRef. If the call is successful, doSaveUsingFSRef is called; if not, doSaveUsingFSSpec is called.

If the user clicked the Choose button in a Choose a Folder dialog, AEGetNthPtr is called to get the file system specification into a local variable of type FSSpec. When a file system specification describes a directory, as it does in this case, 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. The contents of the fields of this FSSpec structure (the directory name, its parent directory ID, and the volume reference number) are then drawn in the bottom of the front window.

Before exit from the kNavCBUserAction case, NavDisposeReply is called to release the memory allocated for the NavReplyRecord structure.

When the user has clicked a dialog's Cancel button, the kNavCBTerminate message is received in the callBackSelector formal parameter. When this message is received, if a dialog reference has been assigned to the global variable gModalToApplicationNavDialogRef (as it will be in the case of the application-modal Open and Choose a Folder dialogs), the dialog is disposed of and the global variable is assigned NULL. If gModalToApplicationNavDialogRef contains NULL, the window reference received in the callBackUD formal parameter is assigned to the local variable windowRef. (Recall that the window reference for the front window was passed in the fifth parameter of the call to NavCreatePutFileDialog.) A handle to the window's document structure is then retrieved, allowing access to the dialog reference stored in that structure. The dialog is disposed of and the relevant field of the document structure is assigned NULL.

Note that, in Carbon applications, there is no need to respond to the kNavCBEvent message in this event function or the following event function in order to call the application's window updating function.

This is assuming the standard window event handler is installed on the relevant windows, the application registers for the kEventWindowDrawContent event type, and calls its window updating function when that event type is received. The following example is provided for those circumstances in which these conditions are not met:

```
case kNavCBEvent:
  switch(callBackParms->eventData.eventDataParms.event->what)
  {
    case updateEvt:
      windowRef = (WindowRef) callBackParms->eventData.eventDataParms.event->message;
      if(GetWindowKind(windowRef) != kDialogWindowKind)
        doUpdate((EventRecord *) callBackParms->eventData.eventDataParms.event);
      break;
  }
  break;
```

askSaveDiscardEventFunction

askSaveDiscardEventFunction is the event (callback) function pertaining to the Save Changes and Discard Changes alerts. It responds to button clicks in those dialogs.

When the user has clicked one of the dialog's buttons, the kNavCBUserAction message is received in the callBackSelector formal parameter. When this message is received, the first action is to get a handle to the front window's document structure. (Recall that the reference to the front window was passed in the third parameter of the NavCreateAskSaveChangesDialog and NavCreateAskDiscardChangesDialog calls.) The main if block executes only if the modalToWindowNavDialogRef field of the document structure contains a dialog reference.

If the user clicked the Save button in a Save Changes alert, doSaveCommand is called to save the file and execution falls through to the kNavUserActionDontSaveChanges case where doCloseDocWindow is called to close the file, flush the volume, and close down the window.

If the user clicked the Don't Save button in a Save Changes alert, and if the program is running on Mac OS X, doCloseDocWindow is called to close the file, flush the volume, and close down the window. If the program is running on Mac OS 9, the global variable gCloseDocWindow is set to true, causing the doCloseDocWindow call to occur in the function doCreateAskSaveChangesDialog. Before all this occurs, NavDialogDispose is called to dispose of the alert before the window is closed by the call to doCloseDocWindow.

If the user clicked the OK button in a Discard Changes alert, the window's content area is erased and the appropriate 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 InvalWindowRect is called to force a redraw of the window's content region. Just before the InvalWindowRect call, 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. The Discard Changes alert is then disposed of.

If the user clicked the Cancel button in a Save Changes or Discard Changes alert, and if it is a Save Changes alert, the flag gQuittingApplication is set to false. This has the effect of defeating the execution of the last block of code in the function doCloseDocWindow. (Recall that the isAskSaveChangesDialog field of the window's document structure is set to true when such alerts are created.) The alert is then disposed of.

doCopyResources

doCopyResources is called by doWriteFile. 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. If the program is running on Mac OS 8/9 and the file type is TEXT, a 'pnot' resource, together with a 'TEXT' resource created within the function, are also copied. (For 'TEXT' files, previews are automatically created on Mac OS X.)

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 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 copied so as to provide a preview for 'PICT' files in the Open dialog. (Of course, in a real application, the 'pnot' and 'PICT' resource would be created by the application for each separate 'PICT' file.)

If the program is running on Mac OS 8/9 and the file type is 'TEXT', a 'pnot' resource is copied and a 'TEXT' resource is created and copied so as to provide a a preview for 'TEXT' files in the Open dialog. After the 'pnot' resource is copied, a relocatable block is created and 1024 bytes of the text in the TextEdit structure is copied to that block. AddResource turns that arbitrary data in memory into a 'TEXT' resource, assigns a resource type, ID, and name to that resource, and inserts an entry in the resource map for the current resource file (in this case, the resource fork of the TEXT file). UpdateResFile then writes the resource map and data to disk.

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.

SynchroniseFiles.c

doSynchroniseFiles

doSynchroniseFiles is called from doIdle when the installed timer fires (every 15 ticks when a document window is the front window).

A reference to the front non-floating window is obtained. The while loop walks the document window section of 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. (Note that kUserDomain is passed in the vRefNum parameter. On Mac OS 8/9, this is mapped to kOnSystemDisk.)

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.

ChooseAFolderDialog.c

doChooseAFolderDialog

doChooseAFolderDialog, which is called when the user chooses the Choose a Folder Dialog item in the demonstration menu, creates and displays a Choose a Folder dialog.

NavGetDefaultDialogCreationOptions initialises the specified NavDialogCreation Options structure with the defaults. GetIndString retrieves a Pascal string, which is converted to a CFString and assigned to the message field of a NavDialogOptions structure. This will appear immediately below the browser list in the Mac OS 8/9 dialog and above the browser list in the Mac OS X dialog.

The next line ensures that the dialog will be application-modal.

NavCreateChooseFolderDialog creates the dialog and NavDialogRun displays it.