16

FILES

Includes Demonstration Program Files1

Preamble

Reference is made in this chapter to the **Standard File Package**, which displays dialog boxes that allow the user to specify the names and locations of files to be save or opened, and which reports the user's choices to your application.

Navigation Services was introduced with Mac OS 8.5 as an alternative to, and ultimately as a replacement for, the Standard File Package.

Those sections of this chapter which address the matter of Open and Save dialog boxes and the reporting of user choices to your application are oriented towards the Standard File Package. The use of Navigation Services for the same purpose is addressed at Chapter 16B — More On Files — Navigation Services.

The demonstration programs associated with this chapter and Chapter 16B are essentially identical except that the demonstration program Files1 uses the Standard File Package whereas the demonstration program Files2 uses Navigation Services.

Macintosh Files

A **file** is a named, ordered sequence of bytes stored on a Macintosh 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 Macintosh Operating System also uses files for certain purposes. For example, 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

All Macintosh files comprise two **forks**, known as the **data fork** and the **resource fork**. Unlike the bytes stored in the resource fork, the bytes in the data fork do not have to exhibit any particular internal structure. Your application is therefore responsible for interpreting the bytes in the data fork in whatever manner is appropriate.

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

Whether you store specific data in the data fork or the resource fork of a file depends largely on whether that data can usefully be structured as a resource. For example, if you want to store a small number of names and telephone numbers, you can easily define a resource type that pairs each name with its telephone number. You can then read names and corresponding numbers from the resource file by using Resource Manager functions. This approach is convenient because, to retrieve data stored in a resource, you simply specify the resource type and ID. You do not need to know, for example, how may bytes of data are stored in the resource.

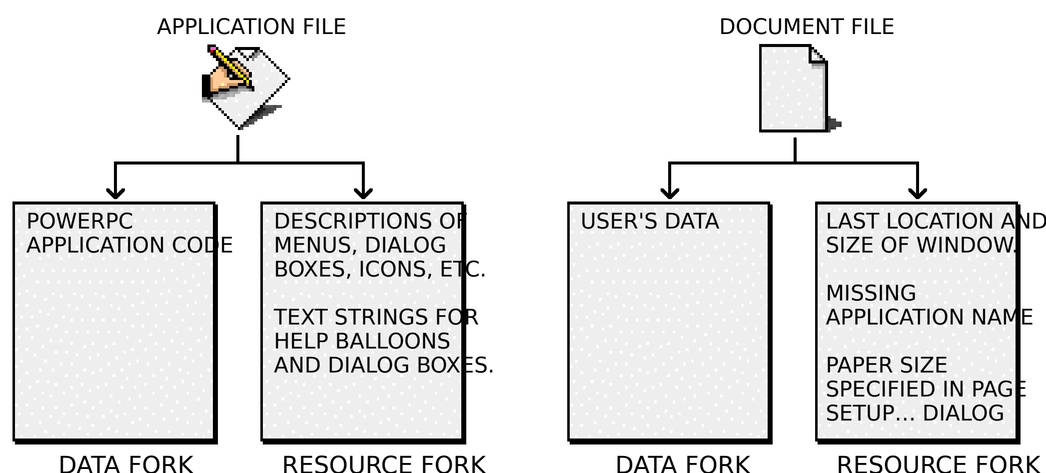

FIG 1 - TYPICAL CONTENTS OF DATA FORKS AND RESOURCE FORKS IN APPLICA

In some cases, however, it is neither possible nor advisable to store your data in resources. For example, it is easiest to store a document's text, which is usually of variable length, in a file's data fork. You can then use File Manager functions to access any byte or group of bytes individually.

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

File Size

Volumes

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

Logical Blocks and Allocation Blocks

The size of a volume varies from one type of device to another. Volumes are formatted into chunks known as **logical blocks**, each of which can contain up to 512 bytes. The actual size of a logical block on a volume is generally only of interest to the disk device driver. This is because the File Manager allocates space to a file in units called **allocation blocks**. An allocation block is a group 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, and optional, Hierarchical File System Plus (HFS+) introduced with Mac OS 8.1.[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. For small volumes, such as volumes on floppy disks, the File Manager uses an allocation block size of only one logical block. However, the larger the volume, the larger is the allocation block. For example, on a 500 MB volume, the allocation block size is 8KB under HFS.

- ***HFS + (Mac OS Extended Format).*** For HFS+-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.

On large volumes, the significant reduction in allocation block size under HFS+ 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+.

Physical and Logical End-Of-File

To distinguish between the amount of space allocated to a file and the number of bytes of actual data in the file, two numbers are used to describe the size of the file:

- ***Physical End-Of-File.*** The physical end-of-file is the number of bytes currently allocated to the file. 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. As a result, the physical end-of-file is always an exact multiple of the allocation block size.

- ***Logical End-Of-File.*** The logical end-of-file is the number of those allocated bytes that currently contain data. It is one greater than the number of the last byte containing data.

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

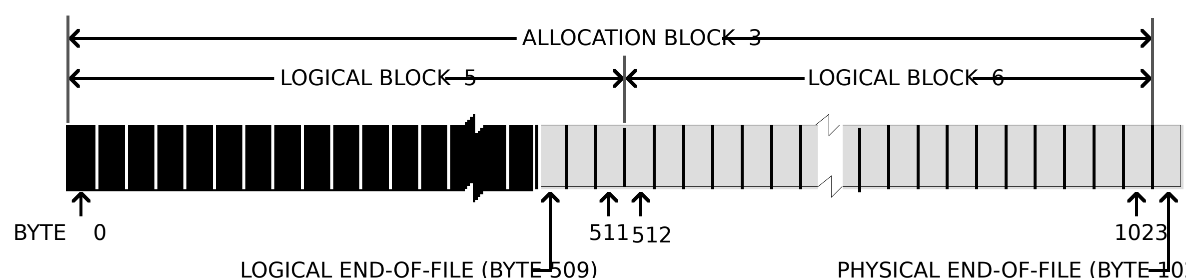

FIG 2 - LOGICAL END-OF-FILE AND PHYSICAL END-OF-FILE

[1] HFS is sometimes referred to as the Mac OS Standard Format. HFS+ is sometimes referred to as the Mac OS Extended Format. The HFS+ format is available for use with any storage device larger than 32MB that support the HFS volume format.

You can move the logical end-of-file to adjust the size of the file. When you move the logical end-of-file to a position more than one allocation block short of the current physical end-of-file, the File Manager automatically deletes the unneeded allocation block from the file. Similarly, if you increase the size of the file by moving the logical end-of-file past the physical end-of-file, the File Manager automatically adds one or more allocation blocks to the file.

Clumps

The number of allocation blocks added to the file is determined by the volume's **clump** size. A clump is a group of contiguous allocation blocks. The purpose of enlarging files by adding clumps is to reduce file fragmentation on a volume, thus improving the efficiency of read and write operations.

Combating File Fragmentation

If you plan to keep extending a file with multiple write operations, and you know in advance approximately how large the file is likely to become, you should first call SetEOF to set the file to that size. This reduces file fragmentation and improves I/O performance.

File Access

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

Access Path and File Reference Number

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

File Mark

For each open access path, the File Manager maintains a current position marker, called the **file mark**, to keep track of where it is in the file during a read or write operation. The mark is the number of the next byte to be read or written. Each time a byte is read or written, the mark is moved. You can specify where each read or write operation should begin by setting the mark or specifying an offset.

Data Buffer

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

Disk Cache

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

The Hierarchical File System

Directories and Directory ID

The Macintosh Operating System uses a method of organising files called an **hierarchical file system**. In an hierarchical file system, files are grouped into **directories** (also called **folders**), which themselves may 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**.

Root Directory

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

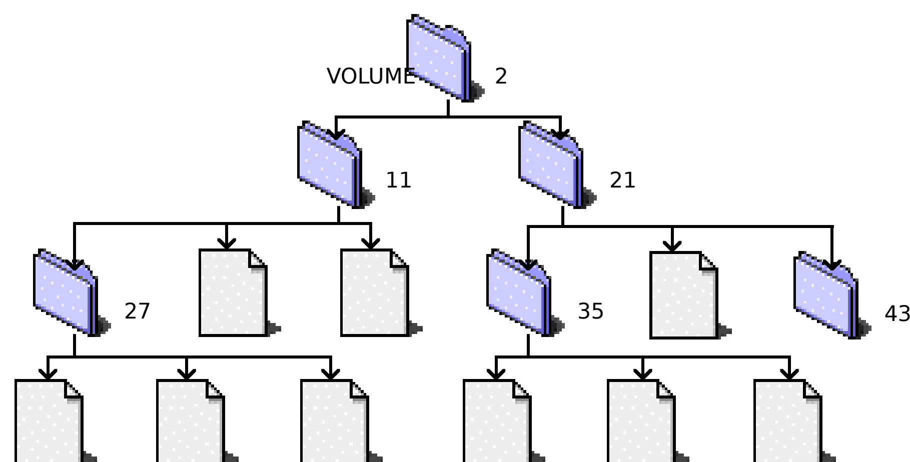

FIG 3 - MACINTOSH HIERARCHICAL FILE SYSTEM

Mounted Volumes

A volume appears on the desktop only after it has been **mounted**. When a volume is mounted, the File Manager places information about the volume in a nonrelocatable block of memory called a **volume control block** (VCB).

When a volume is mounted, the File Manager assigns a **volume reference number** by which you can refer to the volume for as long as it remains mounted. You can also identify a volume by its **volume name**, a sequence of 1 to 27 printing characters (excluding colons)[2]. The volume reference number should be used in preference to the volume name so as to avoid confusion between volumes with the same name.

When an application ejects a disk from a drive, the File Manager places the volume **offline**. When a volume is offline, the volume control block is kept in memory and the volume reference number is still valid. If you make a File Manager call that references that volume, the File Manager presents the disk switch dialog box.

When a user drags a volume icon to the trash, the volume is **unmounted**. The volume control block is released and the volume is no longer known to the File Manager.

[2] The File Manager ignores case when comparing names but does recognize diacritical marks.

Parent Directory and Parent Directory ID

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

For the most part, your application does not need to be concerned about, or keep track of, the location of files in the file system hierarchy. Most of the files your application opens and saves are specified by the user via a dialog box, and their location is provided to your application by either the Finder, the Standard File Package, or Navigation Services. (One notable exception concerns preferences files, which are typically stored in the Preferences folder in the System folder.)

Aliases

In addition to files, folders and volumes, a fourth type of object, namely an **alias**, might appear on the Finder desktop. An alias is a special kind of file which represents another file, folder, or volume. The Finder, the Standard File Package, and Navigation Services automatically resolve aliases.

Identifying Files and Directories

Conventions for identifying files, directories and volumes have evolved as the File Manager has matured. System software Version 7.0 introduced a simple, standard form for identifying a file or directory called the **file system specification**. A file system specification is contained in a structure of type FSSpec:

```
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;
```

In addition to the FSSpec structure, System 7 introduced a new set of high-level functions which accept FSSpec structures as input.

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— Required Apple Events).

The following shows how to perform typical file operations within the context of a user choosing commands from an imaginary application's File menu. 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 from the New, Open…, Close, Save, Save As…, and (optional) Revert to Saved items in the File menu, and for responding to the required Apple events, is illustrated at Fig 4.

Preliminaries - Creating a Document Structure

When a user creates a new document or opens an existing document, your application displays the contents of the document in a window, which provides a standard interface for the user to view, and possibly edit, the document data. It is usual for your application to define a **document structure**, an application-specific data structure which contains information about both the window and the file whose contents are to be displayed in the window.

The following is an example application-defined document structure for an application that handles text files:

```
typedef struct
{
   TEHandle         textEditHdl;      // Handle to TextEdit structure.
   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.
   Boolean          windowTouched;    // Has window's data changed?
}  documentStructure;

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

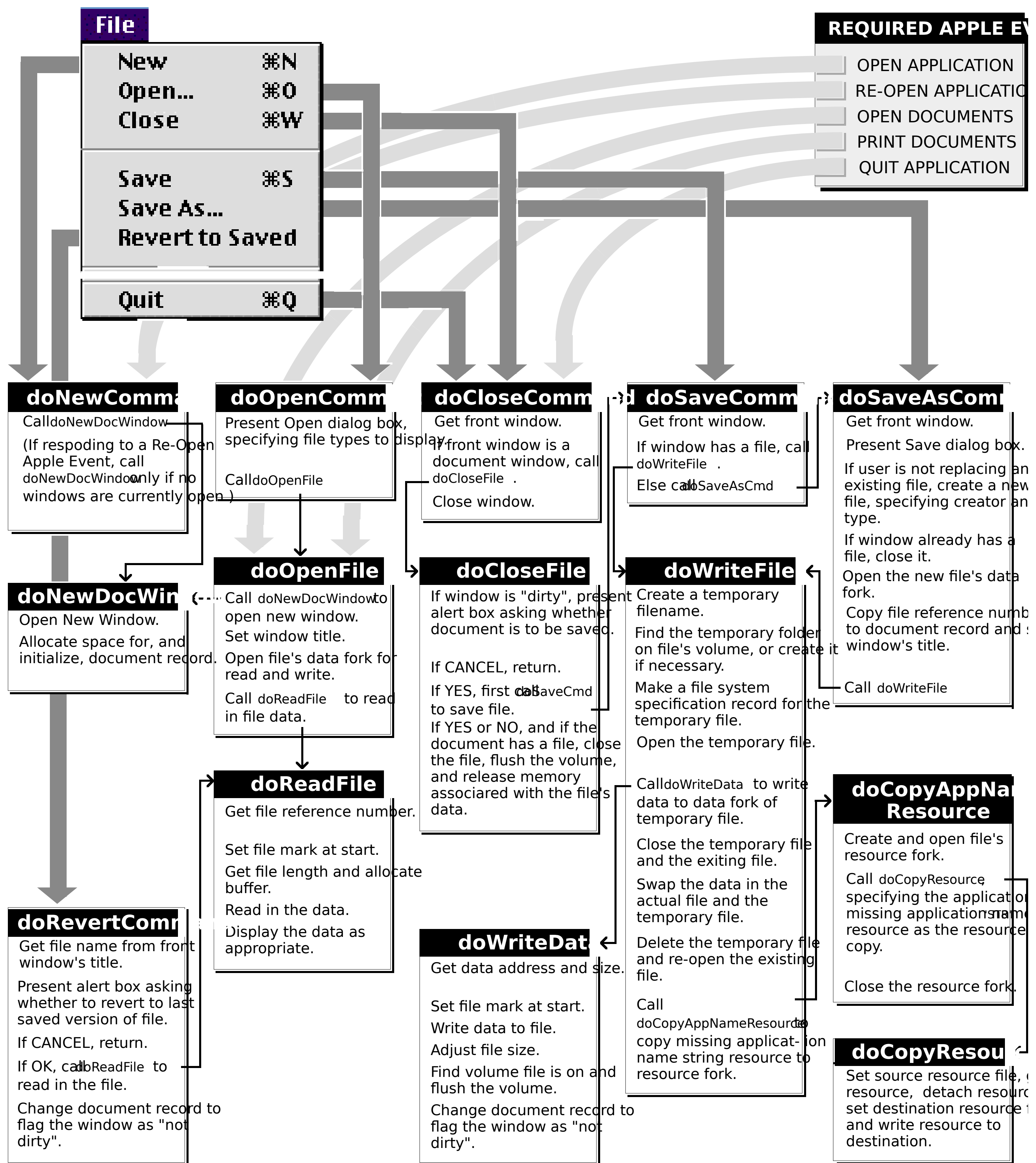

FIG 4 - GENERAL FILE MENU AND REQUIRED APPLE EVENTS HANDLING STRATE

Note the fileRefNum and fileFSSpec fields. Note also that the last field (windowTouched) is used to indicate whether the contents of the document in memory differ from those in the associated file. When your application first reads in the file, it should set this field to false. Then, when any subsequent operations alter the contents of the document in memory, you should set the windowTouched field to true. If the user attempts to close a document window when the value of the windowTouched flag is true, your application should ask the user, via a dialog box, whether to save the changed version of the document to file.

To associate a particular document structure with a particular window, you simply assign the handle to that structure to the reference constant (refCon) field of the window structure using SetWRefCon.

Creating a New Document Window

The user expects to be able to create a new document 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 event from the Finder. Typically, the application-defined function which handles the New... command (doNewCommand at Fig 4) would call another application-defined function (doNewDocWindow at Fig 4), which could be defined along the lines of the following example:

```
OSErr doNewDocWindow(void)
{
    documentStructureHdl docStrucHdl;

    // Open a new window.

    gWindowPtrs[++gNumberOfWindows] = GetNewWindow(rDocWindow,NULL,(WindowPtr) -1);
    if(gWindowPtrs[gNumberOfWindows] == NULL)
        return(MemError());

    // Allocate a relocatable block for a new document structure.

    docStrucHdl = myDocStrucHnd(NewHandle(sizeof(MyDocStruc)));
    if(docStrucHdl == NULL)
    {
        DisposeWindow(gWindowPtr[gNumberOfWindows--]);
        return(MemError());
    }

    // Create new text edit structure. Create scroll bars. Initialise document structure.

    MoveHHi((Handle) docStrucHdl);
    HLock((Handle) docStrucHdl);
    (*docStrucHdl)->textEditHdl = TENew(gDestRect,gViewRect);
    (*docStrucHdl)->vScrollBarHdl = GetNewControl(rVScroll,gWindowPtrs[gNumberOfWindows]);
    (*docStrucHdl)->hScrollBarHdl = GetNewControl(rHScroll,gWindowPtrs[gNumberOfWindows]);
    (*docStrucHdl)->fileRefNum = 0;
    (*docStrucHdl)->windowTouched = false;

    if(((*docStrucHdl)->textEditHdl == NULL) || ((*docStrucHdl)->vScrollBarHdl == NULL)
        || ((*docStrucHdl)->hScrollBarHdl == NULL))
    {
        DisposeWindow(gWindowPtr[gNumberOfWindows--]);
        DisposeControl((*docStrucHdl)->vScrollBarHdl);
        DisposeControl((*docStrucHdl)->hScrollBarHdl);
        TEDispose((((*docStrucHdl)->textEditHdl);
        DisposeHandle((Handle) docStrucHdl);
        return(memFullErr);
    }

    // Make window visible.

    ShowWindow(gWindowPtr[gNumberOfWindows]);

    // Connect document structure to window.

    SetWRefCon(gWindowPtr[gNumberOfWindows],(SInt32) docStrucHdl);

    HUnlock((Handle) docStrucHdl);
    return(noErr);
};
```

Note that this function does not actually create a new file, because it is usually better to wait until the user decides to save the new document before creating a file. Accordingly, doNewDocWindow sets the fileRefNum field of the document structure 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 Print Documents events from the Finder. 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 presenting the standard Open dialog box (see Fig 5).

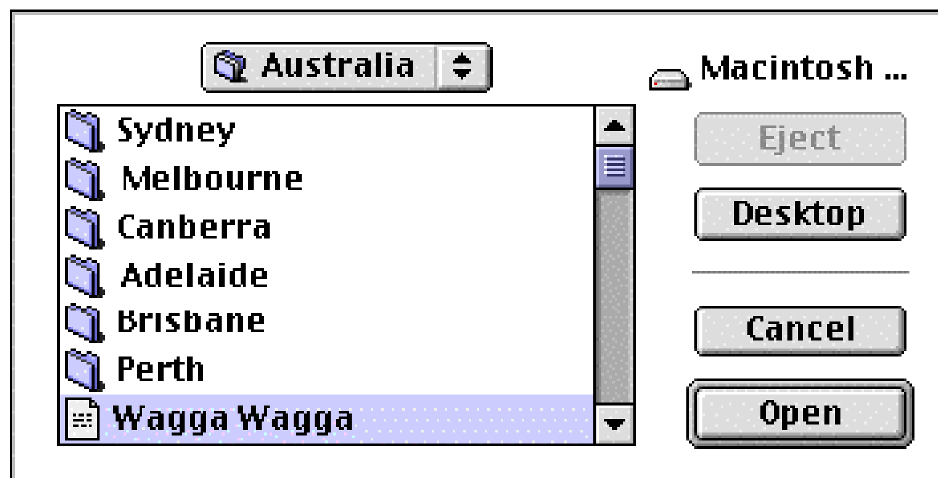

FIG 5 - THE STANDARD OPEN DIALOG BOX

Presenting the Open Dialog Box

StandardGetFile is used to present the standard Open dialog box:

```
void  StandardGetFile(fileFilter,numTypes,typeList,reply);
FileFilterUPP          fileFilter;     Pointer to optional file filter function.
short                  numTypes;       Number of file types to be displayed. -1 = all types.
ConstSFTypeListPtr     typelist;       List of file types to be displayed.
StandardFileReply      *reply;         File reply structure (filled in by StandardGetFile).
```

Standard File Reply Structure. The Open dialog box allows the user to navigate the file system hierarchy and select the required file. While the box is displayed, StandardGetFile handles all events until the user completes the interaction by clicking either the Open button or the Cancel button. When the user clicks one of those buttons, StandardGetFile returns the user's input in the reply parameter, that is, in a Standard File reply structure:

```
struct StandardFileReply
{
   Boolean        sfGood;          // true if user clicked Open button.
   Boolean        sfReplacing;     // true if file to be saved replaces file with same name.
   OSType         sfType;          // File type of the selected file.
   FSSpec         sfFile;          // File system specification for selected item.
   ScriptCode     sfScript;        // Script in which selected item's name is to be displayed.
   short          sfFlags;         // Finder flags of selected item (stationery, etc.).
   Boolean        sfIsFolder;      // true if selected item is a folder.
   Boolean        sfIsVolume;      // true if selected item is a volume.
   long           sfReserved1;     // (Reserved)
   short          sfReserved2;     // (Reserved)
};
typedef struct StandardFileReply StandardFileReply;
```

Creating the Window and Opening the File

If the user clicked the Open dialog box's Open button, the next step is to call the application-defined function (doNewDocWindow at Fig 4) which creates a window and associated document structure and then open the 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 StandardGetFile 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

Once you have opened a file, you can read data from it. Generally, you need to read data from a file when the user first opens it or when the user reverts to the last saved version of a document by choosing the Revert to Saved item in the File menu (see doReadFile at Fig 4). Typically, an application-defined 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.

- Determine 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.
  ```

- Call 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

There are several ways for the user to indicate that the current contents of a document should be saved. The user can choose the File menu commands Save or Save As… or the user can click the Save button in a dialog box that you display when the user attempts to close a "touched" document (that is, a document whose contents have changed since the last time it was saved) (see doCloseCommand at Fig 4). The dialog box used in this latter case would also be presented on receipt of a Quit Application event from the Finder when a "touched" document 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 application-defined function for writing files to disk (see doWriteFile at Fig 4). If the window does not have a file, call the application-defined 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:

- Call StandardPutFile to display the standard Save dialog box (see Fig 6):

```
void  StandardPutFile(prompt,defaultName,reply);
ConstStr255Param     prompt;         Prompt message.
ConstStr255Param     defaultName;    Initial name of file.
StandardFileReply    *reply          File reply structure.
```

  StandardPutFile handles all user interaction until the user clicks the Save or Cancel button. When the user clicks the Open or Cancel button, StandardPutFile returns the user's input in the reply parameter (that is, in a Standard File reply structure).

  If the user clicks on the Save button, perform the remaining steps, otherwise return to the calling function.

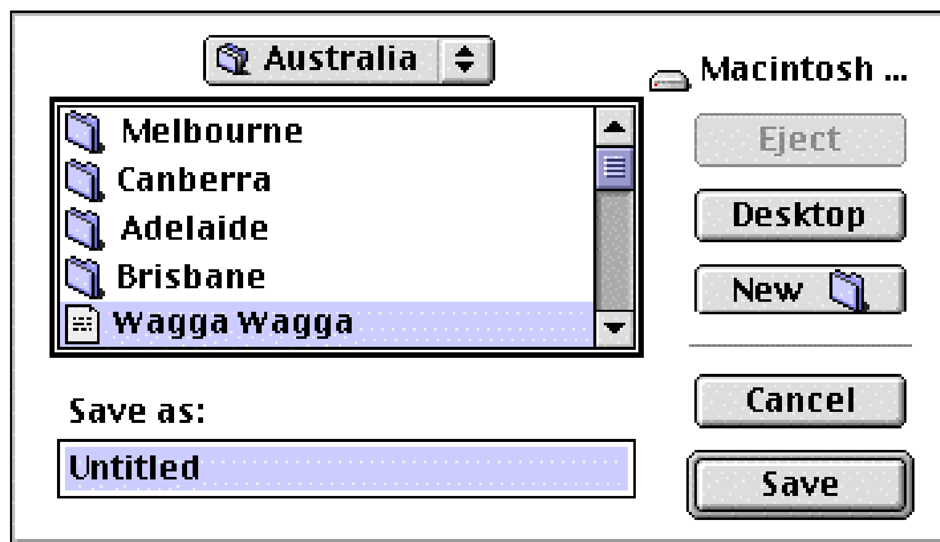

FIG 6 - THE STANDARD SAVE DIALOG BOX

- If the sfReplacing field of the Standard File reply 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.
```

- Copy the sfFile field of the Standard File reply structure to the file system specification structure field of the 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 document structure.

- Call SetWTitle to set the window's title, using the string extracted from the name field of the sfFile field of the Standard File reply structure.

- Call the application-defined function for writing files to disk (see doWriteFile at Fig 4).

Writing File Data

The application-defined 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 application-defined 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[3] are updated.

- Call FSClose to close the temporary file.
- Call FSClose to close the existing file.
- Call FSpExchangeFiles to swap[4] the contents of the temporary file and the existing file:

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

- 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, you should call an application-defined function which copies the missing application name string resource[5] 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);
```

[3] 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.

[4] 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).

[5] See Chapter 9 — Finder Interface.

```
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 an application-defined 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

Many applications that manipulate files provide a Revert to Saved command in the File menu which allows the user to revert to the last saved version of a document. The procedure for handling this command (see doRevertCommand at Fig 4) is relatively simple. You firstly display an alert box asking whether to revert to the last saved version of the file (see Fig 7). If the user clicks the Cancel button, nothing should happen to the current document. If, however, the user clicks on the OK box, you simply call your application-defined function for reading file data (doReadFile at Fig 4) to read the disk version of a file back into the window.

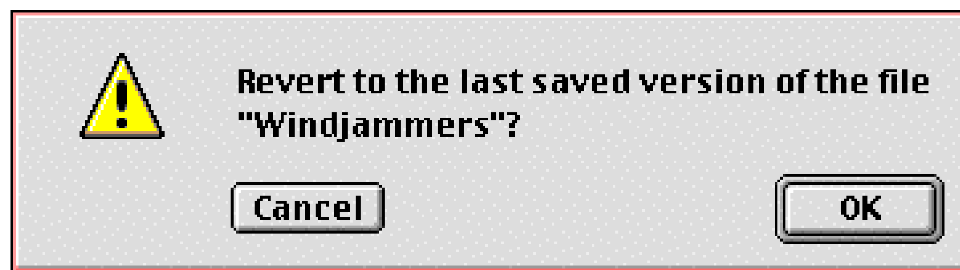

FIG 7 - A REVERT-TO-SAVED DIALOG BOX

Closing a File

Your application must 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 event is received from the Finder.

After determining that the subject window is a document window and not a modeless dialog box (see doCloseCommand at Fig 4), your application should call the application-defined function for closing files (see doCloseFile at Fig 4). This function should:

- Check whether the window is "touched" (that is, whether the contents of the window have been changed since the last time it was saved) by checking the windowTouched field of the document structure.

  If the document has been changed, present the user with a dialog box containing Yes, No and Cancel buttons and asking whether the document should be saved before it is closed. If the user clicks on the Cancel button, simply return. If the user clicks the Yes button, call the application-defined function for saving files and then proceed to the next step. If the user clicks the No button, simply proceed to the next step.

- If the document structure indicates that a file has previously been opened for the document (that is, the file reference number field of the document structure contains a non-zero value), call FSClose to close the file, call FlushVol to ensure that both

the file's data and the file's catalog entry are updated, and set the file reference number field in the document structure to 0.

- Release memory associated with the storage of the file's data. Then dispose of the document structure and, finally, the window.

Customised Open and Save Dialog Boxes

The standard user interfaces provided by StandardGetFile and StandardPutFile may not be adequate for the needs of some applications. To accommodate such cases, the Standard File Package supports customised dialog boxes and, through callback functions, the handling of user actions within customised dialogs.

Typical Reasons for Customising the Standard Dialog Boxes

Specifying File Formats and File Types

A typical reason for customising the Save dialog box would be to allow the user to save a document in one of two file formats. In this case, you might simply add two radio buttons to the standard Save dialog box, as shown at Fig 8. If the application supported a number of different file formats, the radio buttons at Fig 8 could be replaced by a pop-up menu.

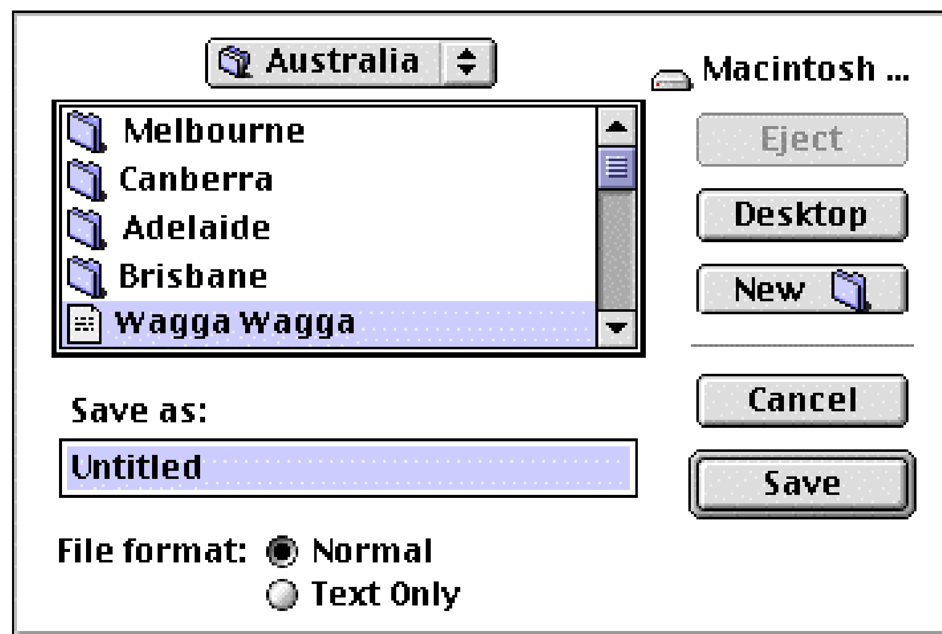

FIG 8 - A CUSTOMISED SAVE DIALOG BOX

A typical reason for customising the Open dialog box is to avoid clutter in the list of files and folders by filtering out all but one of those types. In this case, radio buttons or a pop-up menu might be added to the dialog box to enable the user to select which types of files to view in the list.

Selecting Volumes and Directories

In some circumstances, you need to allow the user to select a volume or directory. For example, the user might want to select a directory as a first step to searching all files in that directory for some specified information. Similarly, the user might want to select a volume before backing up all files on that volume. The standard Open dialog box is, however, designed for selecting files, not volumes or directories. It provides no obvious mechanism for *choosing* a selected directory instead of simply *opening* that directory.

To allow a user to select a directory — including the volume's root directory (the volume itself) — you can add an additional Select push button to the standard Open dialog box together with a Select a Folder: prompt at the top of the dialog box. By clicking the Select push button, the user would be able to select a highlighted directory rather than open it. Fig 9 shows the standard Open dialog box customised to allow the selection of a directory.

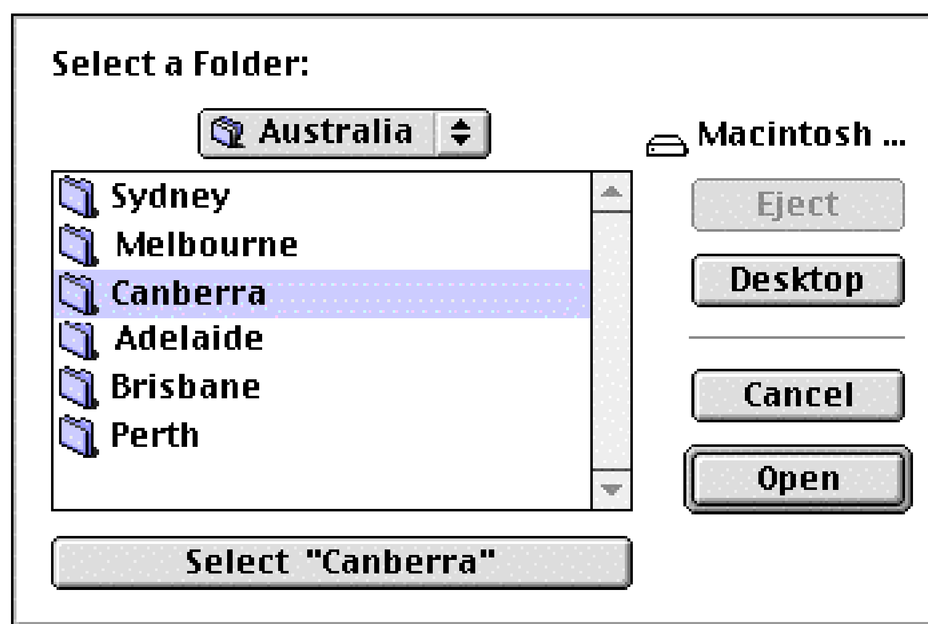

FIG 9 - AN OPEN DIALOG BOX CUSTOMISED TO ALLOW SELEC

Customising the Standard Dialog Boxes

To customise a dialog box, you should:

- Using Resorcerer, copy the 'DLOG' and associated 'DITL' resources for the relevant standard dialog box from the System resource file to your project's resource file, assigning them the required new resource ID number. (The resource IDs for the standard Open and Save dialog boxes in the System file are, respectively, -6042 and -6043.)

- Using Resorcerer, enlarge the dialog box as required to accommodate the additional items and then add the required additional items to the 'DITL' resource, taking care not to change the item numbers of the existing items. (The item numbers for your first additional item will be 10 for the Open dialog box and 13 for the Save dialog box.)

- Write the supporting functions. Depending on the level of customising you require in your dialog box, you may need to write as many as four callback functions:

  - For customised Open dialogs, a **file filter function** for controlling the file types files that will appear in the dialog's list and thus the file types the user can open. For an Open dialog box customised to allow the selection of a directory, a file filter function which causes only folders and volumes to appear in the dialog's list.

  - A **dialog hook function** for handling user actions in the dialog box.

  - A modal dialog filter function for handling user events received from the Event Manager.

  - An activation function for highlighting the display when keyboard input is directed at a customised field defined by your application.

- Call CustomGetFile or CustomPutFile, passing the resource IDs of the customised dialog boxes and universal procedure pointers to the callback functions:

```
void  CustomGetFile(fileFilter,numTypes,typeList,reply,dlgID,where,dlgHook,
                    filterProc,activeList,activateProc,yourDataPtr);

FileFilterYDUPP           fileFilter;      Optional file filter function.
short                     numTypes;        Number of file types to be displayed.
ConstSFTypeListPtr        typeList;        List of file types to be displayed.
StandardFileReply         *reply;          Standard File reply structure.
short                     dlgID;           Dialog resource ID.
Point                     where;           Upper left corner of dialog box (global).
DlgHookYDUPP              dlgHook;         UPP to dialog hook function.
ModalFilterYDUPP          filterProc;      UPP to modal-dialog filter function.
ActivationOrderListPtr    activeList;      Pointer to list of active dialog items.
ActivateYDUPP             activateProc;    UPP to activation function.
void                      *yourDataPtr;    UPP to optional data.

void  CustomPutFile(prompt,defaultName,reply,dlgID,where,dlgHook,
                    filterProc,activeList,activateProc,yourDataPtr);

ConstStr255Param          prompt;          Message to be displayed over text field.
ConstStr255Param          defaultName;     Initial name of file.
StandardFileReply         *reply;          Standard File reply structure.
short                     dlgID;           Dialog resource ID.
Point                     where;           Upper left corner of dialog box (global).
DlgHookYDUPP              dlgHook;         UPP to dialog hook function.
ModalFilterYDUPP          filterProc;      UPP to modal-dialog filter function.
ActivationOrderListPtr    activeList;      Pointer to list of active dialog items.
ActivateYDUPP             activateProc;    UPP to activation function.
void                      *yourDataPtr;    UPP to optional data.
```

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 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, Creating and Deleting 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   FSpDelete(const FSSpec *spec);
```

Exchanging Data in Two Files

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

Creating File System Specifications

```
OSErr   FSMakeFSSpec(short vRefNum,long dirID,ConstStr255Param fileName,FSSpec *spec);
```

Updating Volumes

```
#define PBFlushVol(pb, async) ((async) ? PBFlushVolAsync(pb) : PBFlushVolSync(pb))
```

Obtaining Volume Information

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

Main Standard File Package Data Types and Functions

Data Types

```
typedef const OSType *ConstSFTypeListPtr; // Pointer to an array of OSTypes.
```

Standard File Reply Structure

```
struct StandardFileReply
{
    Boolean       sfGood;          // true if user clicked Open button.
    Boolean       sfReplacing;     // true if file to be saved replaces file with same name.
    OSType        sfType;          // File type of the selected file.
    FSSpec        sfFile;          // File system specification for selected item.
    ScriptCode    sfScript;        // Script in which selected item's name is to be displayed.
    short         sfFlags;         // Finder flags of selected item (stationery, etc).
    Boolean       sfIsFolder;      // true if selected item is a folder.
    Boolean       sfIsVolume;      // true if selected item is a volume.
    long          sfReserved1;     // (Reserved)
    short         sfReserved2;     // (Reserved)
};
typedef struct StandardFileReply StandardFileReply;
```

Functions

Saving Files

```
void    StandardPutFile(ConstStr255Param prompt,ConstStr255Param defaultName,
        StandardFileReply *reply);

void    CustomPutFile(ConstStr255Param prompt,ConstStr255Param defaultName,StandardFileReply
        *reply,short dlgID,Point where,DlgHookYDUPP dlgHook,ModalFilterYDUPP filterProc,
        ActivationOrderListPtr activeList,ActivateYDUPP activate,void *yourDataPtr);
```

Opening Files

```
void    StandardGetFile(FileFilterUPP fileFilter,short numTypes,ConstSFTypeListPtr typeList,
        StandardFileReply *reply);

void    CustomGetFile(FileFilterYDUPP fileFilter,short numTypes,ConstSFTypeListPtr typeList,
        StandardFileReply *reply,short dlgID, Point where,DlgHookYDUPP dlgHook,
        ModalFilterYDUPP filterProc,ActivationOrderListPtr activeList,ActivateYDUPP activate,
         void *yourDataPtr)
```

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)
```

Demonstration Program

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// Files1.h
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
//
// This program demonstrates:
//
// •   Application-defined file handling functions associated with:
//
//     •     The user invoking the File menu Open..., Close, Save..., Save As..., Revert to
//           Saved, and Quit commands of a typical application.
//
//     •     Handling of the required Apple events Open Application, Open Documents and
//           Quit Application.
//
// •   A customised Open dialog which allows the user, via an added pop-up menu button, to
//     specify the types of file (TEXT or PICT) to be displayed in the dialog's list.
//
// •   A directory selection dialog.
//
// 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 making a
// window "touched" (that is, modifying the contents of the associated document).
// Choosing the first menu item in this menu sets the windowTouched 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 are
// "touched".
//
// The program utilises the following resources:
//
```

```
// •   An 'MBAR' resource, and 'MENU' resources for Apple, File, Edit and Demonstration
//     menus (preload, non-purgeable).
//
// •   A 'WIND' resource (purgeable) (initially not visible).
//
// •   Two 'ALRT' resources (purgeable) and associated 'alrx', 'DITL', and 'dftb'
//     resources (purgeable). The first is used to support   the Revert to Saved menu item.
//     The second is used when an attempt is made to close  a modified document before
//     that document has been saved.
//
// •   Two 'DLOG' resources ((purgeable) and associated 'dlgx' and 'DITL' resources
//     (purgeable). The first is for the customised Open dialog.  The second is for the
//     directory selection dialog.
//
// •   Two 'CNTL' resources (purgeable) for the items added to the customised Open dialog.
//     The first is for a pop-up menu button.  The second is for a separator line.
//
// •   A 'MENU' resource for the popu-up menu button.
//
// •   A 'STR ' resource (purgeable) containing the "missing application name" string
//     resource, which is copied to all document files created by the program.
//
// •   A 'STR#' resource containing error strings.
//
// •   A 'SIZE' resource with the acceptSuspendResumeEvents, isHighLevelEventAware, and
//     is32BitCompatible flags set (non-purgeable).
//
// •   The 'BNDL' resource (non-purgeable), 'FREF' resources (non-purgeable), signature
//     resource (non-purgeable), and icon family resources (purgeable), required to
//     support the built application.
//
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

//
.......................................................................................................................................................................
........................................ includes

#include <Appearance.h>
#include <AERegistry.h>
#include <Devices.h>
#include <Folders.h>
#include <Navigation.h>
#include <Resources.h>
#include <Sound.h>
#include <ToolUtils.h>

//
.......................................................................................................................................................................
........................................ typedefs

typedef struct
{
    TEHandle      editStrucHdl;
    PicHandle     pictureHdl;
    SInt16        fileRefNum;
    FSSpec        fileFSSpec;
    Boolean           windowTouched;
}  docStructure, *docStructurePointer, **docStructureHandle;

typedef StandardFileReply *standardFileReplyPtr;

//
.......................................................................................................................................................................
........................................... defines

#define mApple                              128
#define  iAbout                         1
#define mFile                               129
#define  iNew                               1
#define  iOpen                          2
#define  iClose                         4
#define  iSave                          5
#define  iSaveAs                        6
#define  iRevert                        7
#define  iQuit                          12
#define mDemonstration                  131
#define  iTouchWindow                       1
#define  iSelectDirectoryDialog    3
#define rNewWindow                          128
```

```
#define rMenubar                    128
#define rRevertAlert                128
#define rCloseFileAlert             129
#define rCustomOpenDialog           130
#define  iPopupItem                 10
#define rSelectDirectoryDialog 131
#define  iSelectButton              10
#define rErrorStrings               128
#define  eInstallHandler            1000
#define  eMaxWindows                1001
#define  eFileIsOpen                opWrErr
#define kMaxWindows                 10
#define kUserCancelled              1002
#define MAXLONG                     0x7FFFFFFF
#define MIN(a,b)                    ((a) < (b) ? (a) : (b))

// ..........................................................................................................................................................................
......... function prototypes

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

OSErr   doNewCommand            (void);
OSErr   doOpenCommand           (void);
OSErr   doCloseCommand          (void);
OSErr   doSaveCommand           (void);
OSErr   doSaveAsCommand         (void);
OSErr   doRevertCommand         (void);
OSErr   doQuitCommand           (void);
OSErr   doNewDocWindow          (Boolean,OSType);
OSErr   doOpenFile              (FSSpec,OSType);
OSErr   doReadTextFile          (WindowPtr);
OSErr   doReadPictFile          (WindowPtr);
OSErr   doCloseFile             (WindowPtr,docStructureHandle);
OSErr   doWriteFile             (WindowPtr);
OSErr   doWriteTextData         (WindowPtr,SInt16);
OSErr   doWritePictData         (WindowPtr,SInt16);
OSErr   doCopyAppNameResource   (WindowPtr);
OSErr   doCopyResource          (ResType,SInt16,SInt16,SInt16);

pascal  Boolean         filterFunctionOpenDialog     (CInfoPBPtr,void *);
pascal  SInt16          hookFunctionOpenDialog       (SInt16,DialogPtr,void *);
StandardFileReply       doDirectorySelectionDialog   (void);
pascal  Boolean         filterFunctionDirSelect      (CInfoPBPtr,void *);
pascal  SInt16          hookFunctionDirSelect        (SInt16,DialogPtr,void *);

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// Files1.c
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

// ..........................................................................................................................................................................
......................................... includes

#include "Files1.h"

// ..........................................................................................................................................................................
................. global variables

Boolean                 gDone;
```

```
Boolean            gInBackground;
AEEventHandlerUPP doOpenAppEventUPP;
AEEventHandlerUPP doReopenAppEventUPP;
AEEventHandlerUPP doOpenDocsEventUPP;
AEEventHandlerUPP doQuitAppEventUPP;
SInt16             gAppResFileRefNum;

extern SInt16      gCurrentNumberOfWindows;
extern Rect        gDestRect,gViewRect;
extern Boolean gDirectorySelectionFlag;

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ main

void  main(void)
{
  Handle       menubarHdl;
  MenuHandle menuHdl;

  //
..........................................................................................................................................................................
......initialise managers

  doInitManagers();

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

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

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

  gAppResFileRefNum = CurResFile();

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

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

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

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

  doInstallAEHandlers();

  //
..........................................................................................................................................................................
........... enter event loop

  eventLoop();
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ eventLoop

void  eventLoop(void)
{
  EventRecord eventStructure;

  gDone = false;

  while(!gDone)
  {
    if(WaitNextEvent(everyEvent,&eventStructure,MAXLONG,NULL))
      doEvents(&eventStructure);
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doInitManagers
```

```
void  doInitManagers(void)
{
   MaxApplZone();
   MoreMasters();

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

   InitCursor();
   FlushEvents(everyEvent,0);

   RegisterAppearanceClient();
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doInstallAEHandlers

void  doInstallAEHandlers(void)
{
   OSErr   osError;

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

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

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

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doEvents

void  doEvents(EventRecord *eventStrucPtr)
{
   SInt8charCode;

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

      case mouseDown:
         doMouseDown(eventStrucPtr);
         break;

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

      case updateEvt:
         doUpdate(eventStrucPtr);
         break;

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

```
            break;
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doMouseDown

void  doMouseDown(EventRecord *eventStrucPtr)
{
    WindowPtr    windowPtr;
    SInt16       partCode;

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

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

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

        case inDrag:
            DragWindow(windowPtr,eventStrucPtr->where,&qd.screenBits.bounds);
            break;

        case inGoAway:
            if(TrackGoAway(windowPtr,eventStrucPtr->where) == true)
                doCloseCommand();
            break;
    }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doUpdate

void  doUpdate(EventRecord *eventStrucPtr)
{
    WindowPtr            windowPtr;
    docStructureHandle   docStrucHdl;
    GrafPtr              oldPort;
    Rect                 destRect;

    windowPtr = (WindowPtr) eventStrucPtr->message;
    docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);

    GetPort(&oldPort);
    SetPort(windowPtr);

    BeginUpdate(windowPtr);

    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);
    }

    EndUpdate((WindowPtr)eventStrucPtr->message);

    SetPort(oldPort);
```

```
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doMenuChoice

void  doMenuChoice(SInt32 menuChoice)
{
  SInt16               menuID, menuItem;
  Str255               itemName;
  SInt16               daDriverRefNum;
  StandardFileReplystdFileReply;
  Rect                 theRect;
  Str255               theString, numberString;

  menuID = HiWord(menuChoice);
  menuItem = LoWord(menuChoice);

  if(menuID == 0)
    return;

  switch(menuID)
  {
    case mApple:
      GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
      daDriverRefNum = OpenDeskAcc(itemName);
      break;

    case mFile:
      doFileMenu(menuItem);
      break;

    case mDemonstration:
      if(menuItem == iTouchWindow)
        doTouchWindow();
      else if(menuItem == iSelectDirectoryDialog)
      {
        stdFileReply = doDirectorySelectionDialog();
        if(FrontWindow())
        {
          SetPort(FrontWindow());
          TextSize(10);
          SetRect(&theRect,0,271,600,300);
          EraseRect(&theRect);
          if(gDirectorySelectionFlag)
          {
            doCopyPString(stdFileReply.sfFile.name,theString);
            doConcatPStrings(theString, "\p   Volume Reference Number: ");
            NumToString((SInt32) stdFileReply.sfFile.vRefNum,numberString);
            doConcatPStrings(theString,numberString);
            doConcatPStrings(theString, "\p   Parent Directory ID: ");
            NumToString((SInt32) stdFileReply.sfFile.parID,numberString);
            doConcatPStrings(theString,numberString);
            MoveTo(10,290);
            DrawString(theString);
          }
        }
      }
      break;
  }

  HiliteMenu(0);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doFileMenu

void  doFileMenu(SInt16 menuItem)
{
  OSErr   osError;

  switch(menuItem)
  {
    case iNew:
      if(osError = doNewCommand())
        doErrorAlert(osError);
      break;

    case iOpen:
      if(osError = doOpenCommand())
        doErrorAlert(osError);
      break;
```

```
      case iClose:
        if((osError = doCloseCommand()) && osError != kUserCancelled)
          doErrorAlert(osError);
        break;

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

      case iSaveAs:
        if(osError = doSaveAsCommand())
          doErrorAlert(osError);
        break;

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

      case iQuit:
        if((osError = doQuitCommand()) && osError != kUserCancelled)
          doErrorAlert(osError);
        if(osError != kUserCancelled)
          gDone = true;
        break;
  }
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doAdjustMenus

void  doAdjustMenus(void)
{
  MenuHandle         menuHdl;
  WindowPtr          windowPtr;
  docStructureHandle docStrucHdl;

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

  menuHdl = GetMenuHandle(mFile);

  if(gCurrentNumberOfWindows > 0)
  {
    menuHdl = GetMenuHandle(mFile);
    EnableItem(menuHdl,iClose);
    if((*docStrucHdl)->windowTouched)
    {
      EnableItem(menuHdl,iSave);
      EnableItem(menuHdl,iRevert);
    }
    else
    {
      DisableItem(menuHdl,iSave);
      DisableItem(menuHdl,iRevert);
    }
    EnableItem(menuHdl,iSaveAs);

    menuHdl = GetMenuHandle(mDemonstration);
    if((*docStrucHdl)->windowTouched == false)
      EnableItem(menuHdl,iTouchWindow);
    else
      DisableItem(menuHdl,iTouchWindow);
  }
  else
  {
    menuHdl = GetMenuHandle(mFile);
    DisableItem(menuHdl,iClose);
    DisableItem(menuHdl,iSave);
    DisableItem(menuHdl,iSaveAs);
    DisableItem(menuHdl,iRevert);
    menuHdl = GetMenuHandle(mDemonstration);
    DisableItem(menuHdl,iTouchWindow);
  }

  DrawMenuBar();
}
```

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doErrorAlert

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

  paramRec.movable         = true;
  paramRec.helpButton      = false;
  paramRec.filterProc      = NULL;
  paramRec.defaultText     = (StringPtr) kAlertDefaultOKText;
  paramRec.cancelText      = NULL;
  paramRec.otherText       = NULL;
  paramRec.defaultButton   = kAlertStdAlertOKButton;
  paramRec.cancelButton    = 0;
  paramRec.position        = kWindowDefaultPosition;

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

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCopyPString

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

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doConcatPStrings

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

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

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doTouchWindow

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

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

  SetPort(windowPtr);

  TextSize(48);
```

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

  (*docStrucHdl)->windowTouched = true;
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doOpenAppEvent

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

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

  return(osError);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doReopenAppEvent

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

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

  return(osError);
}
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doOpenDocsEvent

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

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

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

    ignoreErr = AEDisposeDesc(&docList);
```

```
    }
    else
        doErrorAlert(osError);

    return(osError);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doQuitAppEvent

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

    osError = doHasGotRequiredParams(appEvent);
    if(osError == noErr)
    {
        while(FrontWindow())
        {
            osError = doCloseCommand();
            if(osError != noErr && osError != kUserCancelled)
                doErrorAlert(osError);
            if(osError == kUserCancelled)
                return;
        }
    }

    gDone = true;

    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)
        return(noErr);
    else if(osError == noErr)
        return(errAEParamMissed);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// NewOpenCloseSave.c
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

//
..........................................................................................................................................................................
......................................... includes

#include "Files1.h"

//
..........................................................................................................................................................................
................. global variables

WindowPtr       gWindowPtr;
SInt16          gCurrentNumberOfWindows = 0;
Rect            gDestRect,gViewRect;
SFTypeList      gFileTypes;

extern SInt16       gAppResFileRefNum;

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doNewCommand

OSErr  doNewCommand(void)
{
    OSErr   osError;
    OSType  documentType = 'TEXT';

    osError = doNewDocWindow(true,documentType);
    return(osError);
}
```

```
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doOpenCommand

OSErr  doOpenCommand(void)
{
  StandardFileReply fileReply;
  OSType            documentType;
  OSErr             osError = noErr;
  FileFilterYDUPP   filterFunctionOpenDialogUPP;
  DlgHookYDUPP      hookFunctionOpenDialogUPP;
  Point             dialogLocation;

  filterFunctionOpenDialogUPP = NewFileFilterYDProc((ProcPtr) filterFunctionOpenDialog);
  hookFunctionOpenDialogUPP = NewDlgHookYDProc((ProcPtr) hookFunctionOpenDialog);

  gFileTypes[0] = 'TEXT';
  gFileTypes[1] = 'PICT';

  dialogLocation.v = -1;
  dialogLocation.h = -1;

  CustomGetFile(filterFunctionOpenDialogUPP,2,gFileTypes,&fileReply,rCustomOpenDialog,
                dialogLocation,hookFunctionOpenDialogUPP,NULL,NULL,NULL,NULL);

  DisposeRoutineDescriptor(filterFunctionOpenDialogUPP);
  DisposeRoutineDescriptor(hookFunctionOpenDialogUPP);

  documentType = fileReply.sfType;

  if(fileReply.sfGood)
    osError = doOpenFile(fileReply.sfFile,documentType);

  return(osError);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCloseCommand

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

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

  switch(windowKind)
  {
    case kApplicationWindowKind:
      docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);
      osError = doCloseFile(windowPtr,docStrucHdl);
      if(osError == kUserCancelled)
        return(kUserCancelled);
      else if(osError == noErr)
      {
        DisposeWindow(windowPtr);
        gCurrentNumberOfWindows --;
      }
      break;

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

  return(osError);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doSaveCommand

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

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

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

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

  return(osError);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doSaveAsCommand

OSErr  doSaveAsCommand(void)
{
  WindowPtr           windowPtr;
  docStructureHandle  docStrucHdl;
  StandardFileReply   fileReply;
  OSType              fileType;
  SInt16              fileRefNum;
  OSErr               osError = noErr;

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

  StandardPutFile("\pSave as:","\pUntitled",&fileReply);

  if(fileReply.sfGood)
  {
    if(!(fileReply.sfReplacing))
    {
      if((*docStrucHdl)->editStrucHdl)
        fileType = 'TEXT';
      else if((*docStrucHdl)->pictureHdl)
        fileType = 'PICT';
      osError = FSpCreate(&fileReply.sfFile,'KKKB',fileType,smSystemScript);
      if(osError != noErr)
        return(osError);
    }

    (*docStrucHdl)->fileFSSpec = fileReply.sfFile;

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

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

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

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

  return(osError);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doRevertCommand

OSErr  doRevertCommand(void)
{
  WindowPtr           windowPtr;
  docStructureHandle  docStrucHdl;
  Str255              fileName;
  SInt16              itemHit;
  OSErr               osError = noErr;
```

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

  SetPort(windowPtr);

  GetWTitle(windowPtr,fileName);
  ParamText(fileName,NULL,NULL,NULL);

  SetPort(windowPtr);

  itemHit = CautionAlert(rRevertAlert,NULL);

  if(itemHit == 1)
  {
    EraseRect(&windowPtr->portRect);
    if((*docStrucHdl)->editStrucHdl)
      osError = doReadTextFile(windowPtr);
    else if((*docStrucHdl)->pictureHdl)
    {
      KillPicture((*docStrucHdl)->pictureHdl);
      (*docStrucHdl)->pictureHdl = NULL;
      osError = doReadPictFile(windowPtr);
    }

    (*docStrucHdl)->windowTouched = false;

    InvalRect(&windowPtr->portRect);
  }

  return(osError);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doQuitCommand

OSErr  doQuitCommand(void)
{
  OSErr   osError = noErr;

  while(FrontWindow())
  {
    osError = doCloseCommand();
    if(osError != noErr)
      return(osError);
  }

  return(osError);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doNewDocWindow

OSErr  doNewDocWindow(Boolean showWindow,OSType documentType)
{
  docStructureHandle  docStrucHdl;

  if(gCurrentNumberOfWindows == kMaxWindows)
    return(eMaxWindows);

  if(!(gWindowPtr = GetNewCWindow(rNewWindow,NULL,(WindowPtr)-1)))
    return(MemError());

  SetPort(gWindowPtr);

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

  SetWRefCon(gWindowPtr,(SInt32) docStrucHdl);

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

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

```
    gViewRect = gDestRect;

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

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

    HUnlock((Handle) docStrucHdl);
  }

  if(showWindow)
    ShowWindow(gWindowPtr);

  gCurrentNumberOfWindows ++;

  return(noErr);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doOpenFile

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

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

  SetWTitle(gWindowPtr,fileSpec.name);

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

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

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

  ShowWindow(gWindowPtr);

  return(noErr);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCloseFile

OSErr  doCloseFile(WindowPtr windowPtr,docStructureHandle docStrucHdl)
{
  Str255  fileName;
  SInt16  itemHit;
  OSErr   osError;

  if((*docStrucHdl)->windowTouched)
  {
    GetWTitle(windowPtr,fileName);
    ParamText(fileName,NULL,NULL,NULL);

    itemHit = CautionAlert(rCloseFileAlert,NULL);
    if(itemHit == 2)
      return(kUserCancelled);
```

```
    else if(itemHit == 1)
    {
      if(osError = doSaveCommand())
        return(osError);
    }
  }

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

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

  DisposeHandle((Handle) docStrucHdl);

  return(osError);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doWriteFile

OSErr  doWriteFile(WindowPtr windowPtr)
{
  docStructureHandle docStrucHdl;
  FSSpec             fileSpecActual, fileSpecTemp;
  UInt32             currentTime;
  Str255             tempFileName;
  SInt16             tempFileVolNum, tempFileRefNum;
  SInt32             tempFileDirID;
  OSErr                osError;

  docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);
  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(windowPtr,tempFileRefNum);
    else if((*docStrucHdl)->pictureHdl)
      osError = doWritePictData(windowPtr,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 = doCopyAppNameResource(windowPtr);

  return(osError);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doReadTextFile

OSErr  doReadTextFile(WindowPtr windowPtr)
```

```
{
   docStructureHandle  docStrucHdl;
   SInt16              fileRefNum;
   TEHandle            textEditHdl;
   SInt32              numberOfBytes;
   Handle              textBuffer;
   OSErr               osError;

   docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);
   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)
   {
      MoveHHi(textBuffer);
      HLockHi(textBuffer);
      TESetText(*textBuffer,numberOfBytes,(*docStrucHdl)->editStrucHdl);
      HUnlock(textBuffer);
      DisposeHandle(textBuffer);
   }
   else
      return(osError);

   return(noErr);
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doReadPictFile

OSErr  doReadPictFile(WindowPtr windowPtr)
{
   docStructureHandle  docStrucHdl;
   SInt16              fileRefNum;
   SInt32              numberOfBytes;
   OSErr               osError;

   docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);
   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(WindowPtr windowPtr,SInt16 tempFileRefNum)
{
   docStructureHandle  docStrucHdl;
   TEHandle            textEditHdl;
   Handle              editText;
   SInt32              numberOfBytes;
   SInt16              volRefNum;
   OSErr               osError;

   docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);
   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(WindowPtr windowPtr,SInt16 tempFileRefNum)
{
  docStructureHandle docStrucHdl;
  PicHandle          pictureHdl;
  SInt32             numberOfBytes, dummyData;
  SInt16             volRefNum;
  OSErr              osError;

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCopyAppNameResource

OSErr  doCopyAppNameResource(WindowPtr windowPtr)
{
  docStructureHandle docStrucHdl;
  OSType             fileType;
  OSErr              osError;
  SInt16             fileRefNum;

  docStrucHdl = (docStructureHandle) GetWRefCon(windowPtr);

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

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

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

    if(fileRefNum > 0)
        osError = doCopyResource('STR ',-16396,gAppResFileRefNum,fileRefNum);
    else
        osError = ResError();

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

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

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doCopyResource

OSErr  doCopyResource(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());
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
// FiltersAndHooks.c
// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊

//
.....................................................................................................................................................................
......................................... includes

#include "Files1.h"

//
.....................................................................................................................................................................
.................. global variables

SInt16  gCurrentType = 1;
Str255  gPrevSelectedName;
Boolean    gDirectorySelectionFlag;

extern SFTypeList gFileTypes;

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ filterFunctionOpenDialog

pascal Boolean  filterFunctionOpenDialog(CInfoPBPtr pbPtr,void *dataPtr)
{
    if(pbPtr->hFileInfo.ioFlFndrInfo.fdType == gFileTypes[gCurrentType - 1])
        return false;
    else
        return true;
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ hookFunctionOpenDialog

pascal SInt16  hookFunctionOpenDialog(SInt16 item,DialogPtr theDialog,void *dataPtr)
{
```

```
	SInt16	theType;
	Handle	controlHdl;
	Rect		theRect;

	switch(item)
	{
		case sfHookFirstCall:
			GetDialogItem(theDialog,iPopupItem,&theType,&controlHdl,&theRect);
			SetControlValue((ControlHandle) controlHdl,gCurrentType);
			return sfHookNullEvent;
			break;

		case iPopupItem:
			GetDialogItem(theDialog,iPopupItem,&theType,&controlHdl,&theRect);
			theType = GetControlValue((ControlHandle) controlHdl);
			if(theType != gCurrentType)
			{
				gCurrentType = theType;
				return sfHookRebuildList;
			}
			break;
	}

	return item;
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ doDirectorySelectionDialog

StandardFileReply  doDirectorySelectionDialog(void)
{
	StandardFileReply	stdFileReplyStruct;
	SFTypeList				fileTypes;
	Point						dialogLocation;
	FileFilterYDUPP		filterFunctionDirSelectUPP;
	DlgHookYDUPP			hookFunctionDirSelectUPP;

	filterFunctionDirSelectUPP = NewFileFilterYDProc((ProcPtr) filterFunctionDirSelect);
	hookFunctionDirSelectUPP = NewDlgHookYDProc((ProcPtr) hookFunctionDirSelect);

	gPrevSelectedName[0] = 0;
	gDirectorySelectionFlag = true;
	dialogLocation.v = -1;
	dialogLocation.h = -1;

	CustomGetFile(filterFunctionDirSelectUPP,-1,fileTypes,&stdFileReplyStruct,
						rSelectDirectoryDialog,dialogLocation,hookFunctionDirSelectUPP,NULL,NULL,
						NULL,&stdFileReplyStruct);

	DisposeRoutineDescriptor(filterFunctionDirSelectUPP);
	DisposeRoutineDescriptor(hookFunctionDirSelectUPP);

	return stdFileReplyStruct;
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ filterFunctionDirSelect

pascal Boolean  filterFunctionDirSelect(CInfoPBPtr pbPtr,void *dataPtr)
{
	SInt32	attributes;
	Boolean	result;

	attributes = (SInt32) pbPtr->hFileInfo.ioFlAttrib;
	result = !(BitTst(&attributes,31 - 4));
	return result;
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊ hookFunctionDirSelect

pascal SInt16  hookFunctionDirSelect(SInt16 item,DialogPtr theDialog,void *dataPtr)
{
	SInt16								itemType, width;
	Handle								itemHdl;
	Rect									itemRect;
	Str255								theName, theString =  "\pSelect  '";
	standardFileReplyPtr	stdFileReplyPtr;

	stdFileReplyPtr = (standardFileReplyPtr) dataPtr;

	if(stdFileReplyPtr->sfIsFolder || stdFileReplyPtr->sfIsVolume)
```

```
    {
      doCopyPString(stdFileReplyPtr->sfFile.name,theName);

      if(IdenticalString(theName,gPrevSelectedName,NULL) != 0)
      {
        doCopyPString(theName,gPrevSelectedName);

        GetDialogItem(theDialog,iSelectButton,&itemType,&itemHdl,&itemRect);
        width = (itemRect.right - itemRect.left) - StringWidth("\pSelect  '    ");
        TruncString(width,theName,smTruncMiddle);
        doConcatPStrings(theString,theName);
        doConcatPStrings(theString,"\p'");

        SetControlTitle((ControlHandle) itemHdl,theString);
      }
    }

    if(item == iSelectButton)
      return sfItemCancelButton;
    else if(item == sfItemCancelButton)
      gDirectorySelectionFlag = false;

    return item;
}

// ◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊◊
```

Demonstration Program 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 which are already open, attempting to save files to new files with existing names, making open windows "touched" by choosing the first item in the Demonstration menu item, reverting to the saved versions of files associated with "touched" windows, choosing Quit when "touched" and non-"touched" windows are open, and so on.

- Choose, via the File types pop-up menu button, the file types required to be displayed in the customised Open dialog.

- Choose the Directory Selection Dialog item from the Demonstration menu to display the directory selection dialog (a customised Open dialog), and select a directory using the Select button at the bottom of the dialog. (The name of the selected directory will be drawn in the bottom-left corner of the front window.)

The program may be run from within CodeWarrior to demonstrate responses to the File menu commands and the directory selection 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.

- 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.

- With several documents open, some with "touched" windows, choose Restart or Shut Down from the Finder's Special menu (thus invoking a Quit Application event), noting that, for "touched" windows, an alert box is presented asking the user whether the file should be saved before the shutdown process proceeds.

Files1.h

#typedef

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

The editRec field will be assigned a handle to a TextEdit edit structure ('TEXT' files). The pictureHdl field will be assigned a handle to a Picture structure ('PICT' files). The fileRefNum and fileFSSpec fields will be assigned the file reference number

and the file system specification structure of the file associated with the window. The windowTouched field will be set to true when a window has been made "touched", that is, when the associated document in memory has been modified by the user.

The second type defined will be used in the dialog hook function for the directory selection dialog.

#define

After the usual constants relating to menus, windows, and alert boxes are established, additional constants are established for the customised Open dialog's 'DLOG' resource and its additional item, the directory selection dialog's 'DLOG' resource and its additional item, a 'STR#' resource containing error strings, and three specific error conditions. kMaxWindows is used to limit the number of windows the user can open. kUserCancelled is used when the user clicks the Cancel button of a particular alert box

Files1.c

Files1.c is simply the basic "engine" which supports the demonstration. There is little in this file which has not featured in previous demonstration programs.

Global Variables

gDone controls termination of the main loop and thus of the program. gInBackground relates to foreground/background switching. The next three globals will be assigned universal procedure pointers to the required Apple event handlers. gAppResFileRefNum will be assigned the file reference number of the application's resource fork.

main

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

doInstallAEHandlers

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

doUpdate

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

doMenuChoice

If the second item in the Demonstration menu is chosen, the application-defined function which presents the directory selection dialog is called. This function returns a standard file reply structure. If a window is open, a rectangle in the bottom corner of the front window is erased. If the dialog was dismissed using the Select button (not the Cancel button), the selected directory name, volume reference number, and parent directory ID are extracted from the standard file reply structure and drawn in the bottom of the window.

doFileMenu

doFileMenu handles File menu choices. In each case, the relevant application-defined function is called and, if that function returns an error, the application-defined function doErrorAlert is called. Note that, in the case of the Quit command, gDone is set to true after doQuitCommand returns, thus causing the program to terminate.

doErrorAlert

doErrorAlert handles errors, invoking an appropriate alert box (caution or stop) advising of the nature of the problem by error code number or straight text. Note that he program will only be terminated in the case of the memFullErr error (no more space in the application heap).

doTouchWindow

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

doOpenAppEvent, doOpenDocsEvent, and doQuitAppEvent

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

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

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

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

Within the function doQuitAppEvent, the while loop entered at repeats for each open window. Within the loop, doCloseCommand is called. doCloseCommand, in turn, calls doCloseFile. doCloseFile presents a Yes/No/Cancel caution alert. If an error is returned by this sequence, and if the user did not click the Cancel button in the alert, the error handler is called. If the user clicked the Cancel button, it is necessary to interrupt the sequence of closing all open windows and re-enter the main event loop.

When the while loop eventually exits, gDone is set to true, causing the program to terminate.

NewOpenCloseSave.c

Global Variables

gWindowPtr is assigned the pointer to the graphics port of each new window as it is opened. gCurrentNumberOfWindows keeps a count of the number of windows opened. gDestRect and gViewRect are used to set the destination and view rectangles for the edit structures associated with 'TEXT' files. gFileTypes will control the file types to be displayed in the Open dialog box.

doNewCommand

doNewCommand is the first of the file-handling functions. It is called when the user chooses New from the File menu and when an Open Application 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 immaterial. The document type 'TEXT' is passed in this instance simply to keep doNewDocWindow happy.

doOpenCommand

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

The first two lines create routine descriptors for the file filter and dialog hook functions utilised by the customised Open dialog box. At the next two lines, the first two elements of the gFileTypes are assigned the file types to be displayed. -1 is then assigned to both fields of dialogLocation.

The call to CustomGetFile presents the customised Open dialog box. The first parameter is a UPP to an application-defined file filter function. As will be seen, this filter function will either allow or block the display of one or other of the two file types specified in the first two elements of gFileTypes. The sixth parameter ordinarily specifies the location of the top-left of the dialog box on the screen; however, the assignment of -1 to both fields of dialogLocation will cause CustomGetFile to centre the dialog box on the screen. Note also that the next parameter is a UPP to the application-defined dialog hook function which handles user interaction with the dialog.

When the dialog box is dismissed, the routine descriptors are disposed of.

The sfType field of the StandardFileReply structure "filled-in" by CustomGetFile (fileReply) contains the file type of the file selected by the user and the sfFile field contains the file system specification. If the user clicks the OK button (the third last line), these are passed to the application-defined function doOpenFile.

> To use the standard Open dialog, delete the UPP and Point variables and their associated code, make gFilesType a local variable, and replace the call to CustomGetFile with:
>
> ```
> StandardGetFile(NULL,2,gFileTypes,&fileReply);
> ```

doCloseCommand

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

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

If the front window is a document window, the handle to the window's document structure is retrieved from the window structure's refCon field. The WindowPtr and this handle are then passed to the application-defined function doCloseFile. If the window is "touched", doCloseFile presents an alert box asking the user whether the document should be saved before it is closed. If the user clicks the Cancel button of that alert box, doCloseFile returns kUserCancelled, in which case doCloseCommand returns kUserCancelled. If the user clicks either the Yes or No buttons of the alert box, and if doCloseFile returns no error, the window is closed as the final act in closing the file, and the global variable which keeps track of the number of open windows is decremented.

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

doSaveCommand

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

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

doSaveAsCommand

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

The first two lines get the WindowPtr for the front window and retrieve the handle to that window's document structure.

The call to StandardPutFile presents the Save dialog box. The remaining code executes only if the user clicks on the Save button.

If the sfReplacing field of the StandardFileReply structure "filled-in" by StandardPutFile indicates that an existing file is not being replaced, the file type is retrieved from the document structure for the front window and FSpCreate is called to create a new file of that type, specifying the application's signature as the creator.

The file system specification structure returned in the sfFile field of the StandardFileReply structure tis then assigned to the fileFSSpec field of the document structure.

If a file currently exists for the document, that file is closed by the call to FSClose.

The data fork of the newly created file is then opened by a call to FSpOpenDF, the fileRefNum field of the document structure is assigned the file reference number returned by FSpOpenDF, the window's title is set to the new file's name, and the application-defined function doWriteFile is called to write the document to the new file.

doRevertCommand

doRevertCommand is called when the user chooses Revert to Saved from the File menu.

The first three lines get the WindowPtr for the front window, retrieve the handle to that window's document structure, and retrieve the file reference number from the document structure.

The call to GetWTitle gets the window's title (that is, the filename) for insertion by ParamText into the text of the alert box invoked by the call to CautionAlert. (The alert box asks the user to confirm, or otherwise, the reversion to the last saved version.)

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

doQuitCommand

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

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

doNewDocWindow

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

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

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

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

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

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

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

doOpenFile

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

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

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

doCloseFile

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

If the window is touched, a caution alert is presented asking the user whether the document should be saved. (GetWTitle and ParamText insert the window title into the text in the alert box.) The alert box contains Yes, No and Cancel buttons. If the user clicks Cancel, the function returns kUserCancelled. If the user clicks Yes, the application-defined function doSaveCommand() is called to save the file.

If the user clicks Yes or No, the next block executes:

- If the document has a file, FSClose closes the file, and FlushVol stores to disk all unwritten data currently in the volume buffer.
- If the document is a text document, the text edit structure is disposed of. If it is a picture document, the Picture structure is disposed of. Finally, the document structure is disposed of.

doWriteFile

doWriteFile is called by doSaveCommand and doSaveAsCommand. In conjunction with two supporting application-defined functions, it writes the document to disk 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 application-defined 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 by changing the information in the volume's catalog. The temporary file is then deleted and the data fork of the existing file is re-opened.

The application-defined function doCopyAppNameResource 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.

doReadTextFile

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

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

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

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

NewHandle allocates a buffer equal to the size of the file (or 32,767 bytes if the preceding if statement executed). FSRead reads the data from the file into the buffer. MoveHHi and HLockHi move the buffer high in the heap and lock it preparatory to the call to 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.

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

doReadPictFile

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

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

doWriteTextData

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

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

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

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

doWritePictData

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

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

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

doCopyAppNameResource

doCopyAppNameResource is called by doWriteFile when a newly created file has been written to for the first time. It copies the missing application name string resource from the resource fork of the application file to the resource fork of the new file.

The first line retrieves a handle to the file's document structure. The next four lines establish the file type involved. FSpCreateResFile creates the resource fork in the new file and FSpOpenResFile opens the resource fork. The application-defined function for copying specified resources between specified files (doCopyResource) is then called. In this case, the specified resource is the missing application name string resource, the source resource file is the resource fork of the application file, and the destination resource file is the resource fork of the new file.

CloseResFile closes the resource fork of the new file.

doCopyResource

doCopyResource copies specified resources between specified files. In this program, it is called only by doCopyAppNameResource.

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.

FiltersAndHooks.c

FiltersAndHooks.c contains the file filter and dialog hook functions for the customised Open dialog and the directory selection dialog, together with the function which calls up the directory selection dialog.

filterFunctionOpenDialog

filterFunctionOpenDialog is the file filter function for the customised Open dialog.

The global variable gCurrentType contains an index into the gFileTypes array (see the function doOpenCommand, above). As will be seen, the index changes according to the item chosen by the user using the two-item pop-up menu button added to the dialog box.

If the file type passed in for evaluation matches the current file type indexed by gFileTypes, the filter returns false, indicating that CustomGetFile should put it in the list. All other file types are blocked as a result of the filter function returning true.

hookFunctionOpenDialog

hookFunctionOpenDialog is the hook function for the customised Open dialog. It handles item selection within the dialog. CustomGetFile calls this function immediately after calling ModalDialog, passing the function the item number returned by ModalDialog.

A dialog hook function is required to return the item number passed to it or some other item number. This includes "psuedo item numbers", that is, constants which do not represent actual items in the item list. There are two types of psuedo items: psuedo items passed to a dialog hook function by the Standard File Package; psuedo items passed to the Standard File Package by a dialog hook function.

The sfHookFirstCall constant (see the first case in the switch statement) is an example of the first kind of psuedo item. This psuedo item is sent to the dialog hook function immediately before the dialog is displayed. The dialog hook function typically reacts to this item by performing any necessary initialisation. In hookFunctionOpenDialog, this initialisation involves setting the value of the pop-up menu button control (that is, the current menu item) to equate to the initial value in the global variable gCurrentType (1).

A dialog hook function can pass back psuedo items to request some action by the Standard File Package, or to indicate that it needs no further action by the Standard File Package. The last action following the receipt of the sfHookFirstCall psuedo item is to return the sfHookNullEvent, which indicates that no further action is required.

At the second case, if the pop-up menu button was hit, the control's value is retrieved and compared with the value currently in the global gCurrentType. If the two do not match, gCurrentType is assigned the current value of the control (that is, the menu item currently chosen) and the psuedo item sfHookRebuildList is returned. This psuedo item instructs the Standard File Package to rebuild the list of files and folders.

If the item passed to hookFunctionOpenDialog was not sfHookFirstCall or the pop-up menu button item, the last line returns that item number to the Standard File Package for processing.

DIALOG HOOK FUNCTIONS FOR CUSTOMISED SAVE DIALOGS

A dialog hook function for a customised Save dialog must account for the fact that user action can result in another dialog opening on top of the Save dialog. For example, if the user hits the New Folder button, the New Folder dialog box will open. CustomPutFile will call your dialog hook function for item selections in subsidiary dialogs as well as the main dialog. Your dialog hook function must therefore return immediately if the dialog pointer received does not pertain to the customised Save dialog itself. The content of the dialog window's refCon field may be used to discriminate between the various dialogs.

The refCon field of a standard Save dialog's window contains sfMainDialogRefCon (1937007718). Accordingly, you can assign that value to your customised dialog window's refCon field and include the following as the beginning of your dialog hook function:

```
if((GetWRefCon((WindowPtr) theDialog)) != (SInt32) sfMainDialogRefCon)
    return;
```

doDirectorySelectionDialog

doDirectorySelectionDialog is called when the user chooses the Directory Selection Dialog item in the Demonstration menu.

The directory selection dialog box is a customised version of the standard Open dialog. It adds a static text item at the top and a "Select" push button immediately below the list.

The first two lines create routine descriptors for the filter and hook functions. Two global variables are then initialised. The first is used to store the selected directory name and the second (gDirectorySelectionFlag) is a flag which the hook function will set to false if the user hits the Cancel button in the dialog box. The values assigned to the fields of the dialogLocation variable ensure that the dialog will be displayed in the centre of the screen.

CustomGetFile displays the dialog. When the user dismisses the dialog, the routine descriptors are disposed of and the standard file reply structure "filled in" by CustomGetFile is returned to the calling function (doMenuChoice), where, if the dialog was dismissed using the Select button (not the Cancel button), the selected directory name, volume reference number, and parent directory ID are extracted and drawn in the bottom of the window.

filterFunctionDirSelect

filterFunctionDirSelect is the filter function for the directory selection dialog. It inspects the appropriate bit in the file attributes field of the catalog information parameter block passed to it. If the directory bit (bit 4) is set, false is returned, indicating that the item should appear in the list; otherwise, true is returned to exclude the item from the list.

hookFunctionDirSelect

hookFunctionDirSelect is the hook function for the directory selection dialog.

No initialisation is required by this hook function. Accordingly, although an sfHookFirstCall psuedo item will be received just before the dialog is displayed, it is disregarded.

If the sfIsFolder or sfIsVolume fields of the standard file reply structure pointed to by dataPtr indicate that the selected item is a folder or a volume, and if the name extracted from the standard file reply structure is not the same as that stored in the global variable gPrevSelectedName, the new name is copied to gPrevSelectedName. Also, the new string for the Select push button's title is built up prior to a call to SetControlTitle. If the string is too wide for the push putton, it is centre-truncated.

If the item hit was the Select push button, a return is forced by faking a cancel (that is, sfItemCancelButton is returned even though the Cancel button was not hit. This will cause the dialog to be dismissed. If, on the other hand, the Cancel button was hit, the global variable gDirectorySelectionFlag is set to false and the last line returns the item number received (sfItemCancelButton), causing the dialog to be dismissed. (In this demonstration, gDirectorySelectionFlag simply allows or defeats the drawing of the selected directory name, volume reference number, and parent directory ID in the bottom of the window.)