14 Version 1.1

FILES

Includes Demonstration Program FilesPascal

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 located in a volume to maintain the hierarchical organisation of files and folders in that volume. This special file is called the volume's **catalog file**.
- If virtual memory is in operation, the Operating System stores unused pages of text in a disk file called the **backing store file**.

Characteristics of Files

File Forks

All Macintosh files comprise two **forks**, known as the **data fork** and the **resource fork**. The resource fork contains the file's resources. The data fork contains the file's data. 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 routines. 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.

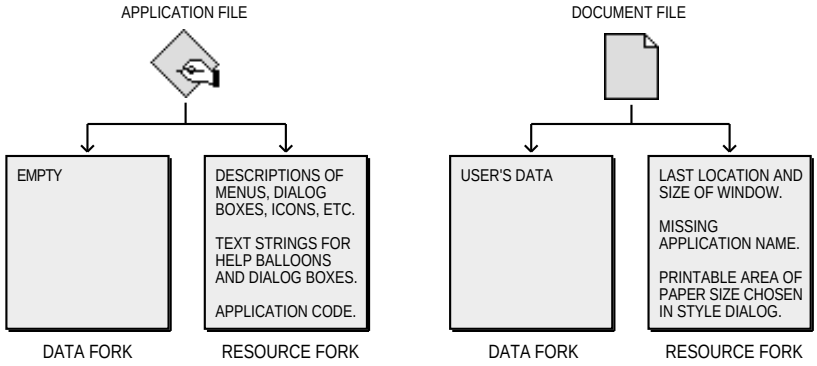

FIG 1 - TYPICAL CONTENTS OF DATA FORKS AND RESOURCE FORKS IN APPLICATION AND DOCUMENT FILES

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 routines 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.

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 one logical block. To support volumes larger than about 32 MB, the File Manager needs to use an allocation block size which is at least two logical blocks. To support volumes larger than about 64 MB, the File Manager needs to use an allocation block size which is at least three logical blocks.

A non-empty file fork always occupies at least one allocation block. On a 40 MB volume, for example, a file's data fork occupies at least 1024 bytes (two logical blocks) even if it contains only, say, 11 bytes of actual data.

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.

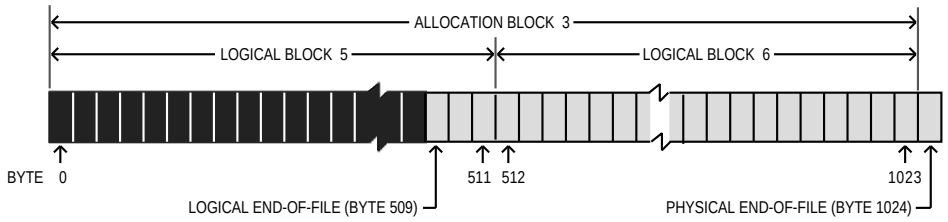

FIG 2 - LOGICAL END-OF-FILE AND PHYSICAL END-OF-FILE

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 the **hierarchical file system** (HFS). In HFS, 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**.

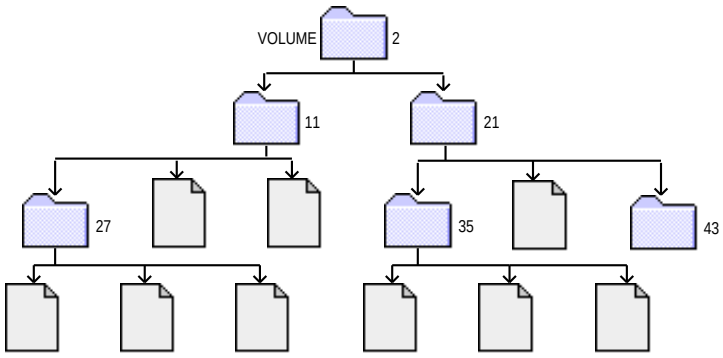

FIG 3 - MACINTOSH HIERARCHICAL FILE SYSTEM

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**.

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)[1]. 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 3.5 inch 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.

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

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.

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 or the Standard File Package. (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 and the Standard File Package 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 record of type `FSSpec`:

```
FSSpec = record
  vRefNum:  integer;
  parID:    longint;
  name:     Str63;
  end;

FSSpecPtr = ^FSSpec;
FSSpecHandle = ^FSSpecPtr;
```

In addition to the `FSSPec` record, System 7 introduced a new set of high-level routines which accept `FSSPec` records as input.

This chapter is concerned only with the routines, and the method of identifying files and directories, introduced with System 7. Older methods of identifying files and directories (file ID references, working directory reference numbers, full or partial pathnames, etc.) are not addressed in the following.

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 8 — 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.

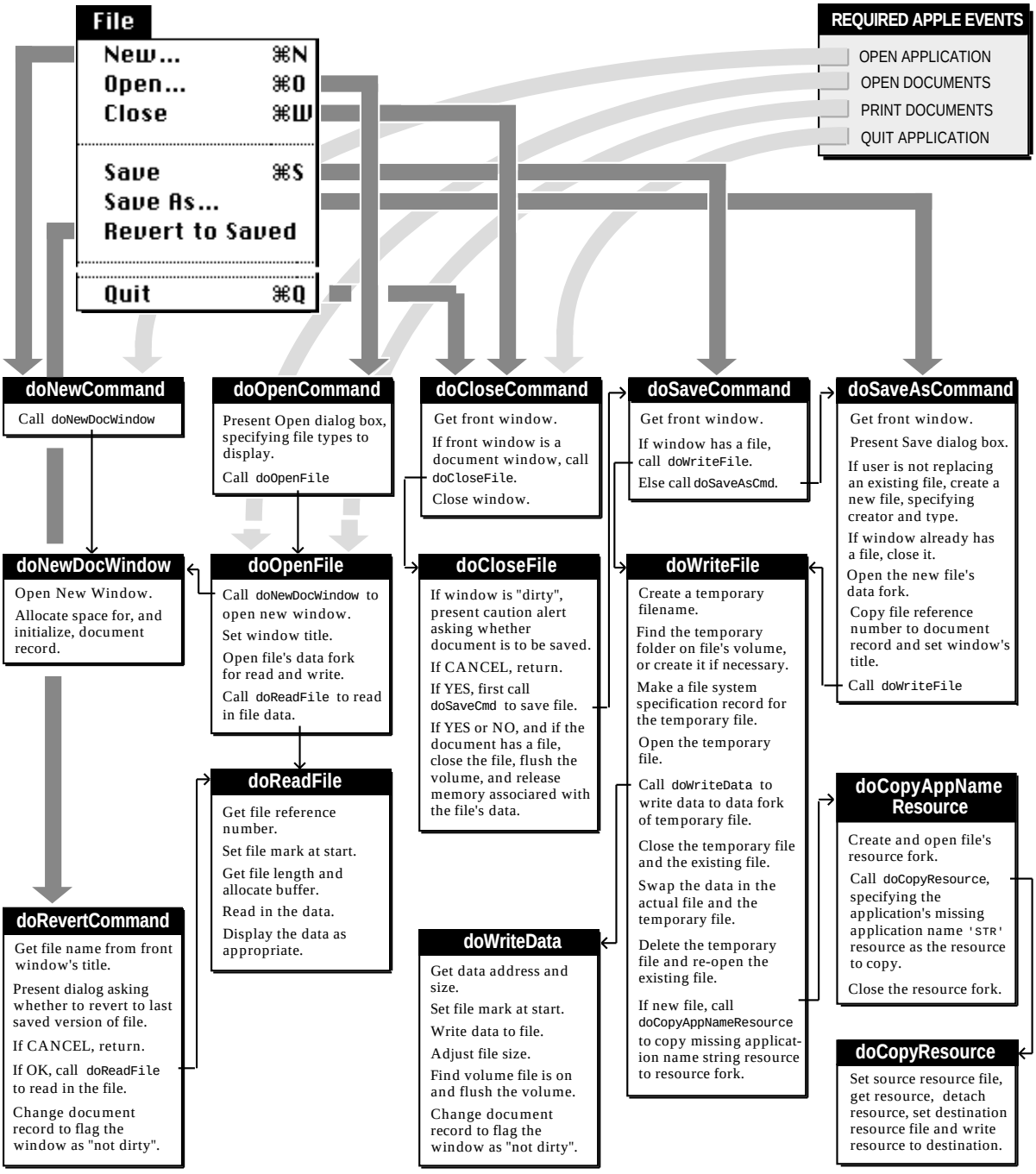

FIG 4 - GENERAL FILE MENU AND REQUIRED APPLE EVENTS HANDLING STRATEGY

Preliminaries - Creating a Document Record

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 record**, 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 record for an application that handles text files:

```
DocumentRecord = record
  editRec : TEHandle;          { Handle to TextEdit record.}
  vScrollBar : ControlHandle;  { Handle to vertical scroll bar.}
  hScrollBar : ControlHandle;  { Handle to horizontal scroll bar.}
  fileRefNum : SInt16;         { File reference number for window's file.}
  fileFSSpec : FSSpec;         { File's file system specification record.}
  windowDirty : Boolean;       { Has window's data changed?}
end;

DocumentRecordPtr = ^DocumentRecord;
DocumentRecordHdl = ^DocumentRecordPtr;
```

Note the `fileRefNum` and `fileFSSPec` fields. Note also that the last field (`windowDirty`) 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 `windowDirty` field to `true`. If the user attempts to close a document window when the value of the `windowDirty` 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 record with a particular window, you simply assign the handle to that record to the reference constant (`refCon`) field of the window record 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:

```
function DoNewDocWindow : OSErr;

var
  docRecHdl : DocumentRecordHdl;

begin
  { Open a new window.}

  gNumberOfWindows := gNumberOfWindows + 1;
  gWindowPtrs[gNumberOfWindows] := GetNewWindow(rDocWindow, NIL, WindowPtr(-1));
  if (gWindowPtrs[gNumberOfWindows] = NIL) then
    DoNewDocWindow := MemError;

  { Allocate a relocatable block for a new document record.}

  docRecHdl := myDocRecHnd(NewHandle(sizeof(MyDocRec)));
  if (docRecHdl = NIL) then
    begin
    gNumberOfWindows := gNumberOfWindows - 1;
    DisposeWindow(gWindowPtr[gNumberOfWindows]);
    DoNewDocWindow := MemError;
    end;

  { Create new text edit record.  Create scroll bars.  Initialise document record.}

  MoveHHi(Handle(docRecHdl));
  HLock(Handle(docRecHdl));
  docRecHdl^^.editRec := TENew(gDestRect, gViewRect);
  docRecHdl^^.vScroll := GetNewControl(rVScroll, windowPtr);
  docRecHdl^^.hScroll := GetNewControl(rHScroll, windowPtr);
  docRecHdl^^.fileRefNum := 0;
  docRecHdl^^.windowDirty := false;
```

```
  if ((editRec = NIL) or (vScroll = NIL) or (hScroll = NIL)) then
    begin
    DisposeWindow(gWindowPtr[gNumberOfWindows--]);
    DisposeControl(vScroll);
    DisposeControl(hScroll);
    TEDispose(editRec);
    DisposeHandle(Handle(docRecHdl));
    DoNewDocWindow := memFullErr;
    end;
  { Make window visible.}

  ShowWindow(gWindowPtr[gNumberOfWindows]);

  { Connect document record to window.}

  SetWRefCon(gWindowPtr[gNumberOfWindows], longint(docRecHdl));

  HUnlock(Handle(docRecHdl));
  DoNewDocWindow := noErr;
end;
```

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 record 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).

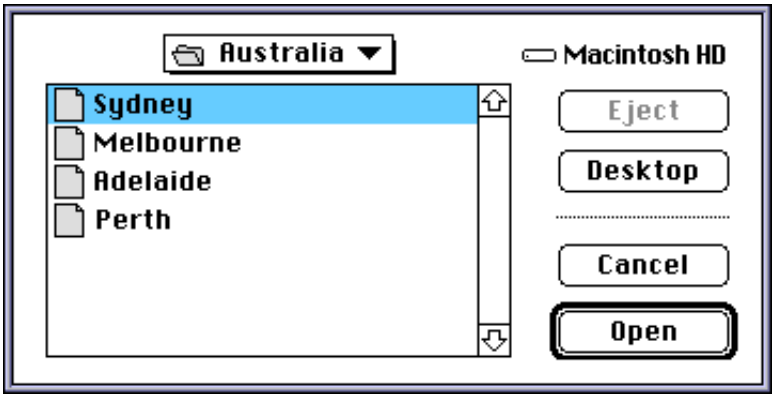

FIG 5 - THE STANDARD OPEN DIALOG BOX

Presenting the Open Dialog Box

`StandardGetFile` is used to present the standard Open dialog box:

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

Standard File Reply Record. 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 record:

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

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 record and then open the file's data fork (`doOpenFile` at Fig 4).

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

```
function FSpOpenDF  (spec,permission,refNum) : OSErr;
spec :       FSSpec;       File system specification record.
permission : SInt8;        Access mode.
refNum :     integer;      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 record so that it can be readily retrieved for use as a parameter in calls to routines 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 record.
- Calls `SetFPos` to set the file mark to the beginning of the file:

```
function SetFPos (refNum,posMode,posOff) : OSErr;
refNum :   integer;   File reference number.
posMode :  integer;   Positioning mode.
posOff :   longint;   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` :

```
function GetEOF (refNum,logEOF) : OSErr;
refNum : integer;    File reference number.
LogEOF : longint;    Receives length of file, in bytes.
```

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

```
function  FSRead (refNum,count,buffPtr) : OSErr;
refNum : integer;    File reference number.
count : longint;     On input: bytes to read.  On output: actual bytes read.
BuffPtr : UNIV Ptr;  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 "dirty" 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 "dirty" 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 record 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):

```
procedure StandardPutFile   (prompt,defaultName,reply);
prompt :      ConstStr255Param;   Prompt message.
defaultName : ConstStr255Param;   Initial name of file.
reply :       StandardFileReply;  File reply record.
```

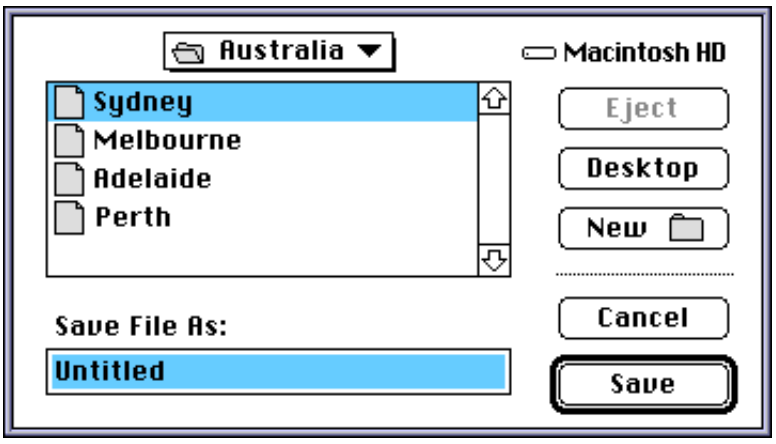

FIG 6 - THE STANDARD SAVE DIALOG BOX

`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 record).

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

- If the `sfReplacing` field of the Standard File reply record does not contain `true`, call `FSpCreate` to create a new file and set the file type and creator:

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

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

- If the window already has a file (that is, if the file reference number field of the document record does not contain 0), close that file with a call to `FSClose`:

```
function FSClose (refNum) : OSErr;
refNum : integer;       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 record.

- 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 record.

- 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 record.

- 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:

```
function FindFolder  (vRefNum,folderType,createFolder,foundVRefNum,foundDirID) :
          OSErr;

vRefNum :       integer;  Volume reference number.
folderType :    OSType;   Folder type for volume.
createFolder :  boolean;  kCreateFolder or kDontCreateFolder.
foundVRefNum :  integer;  Volume reference number for folder found.
foundDirID :    longint;  Directory ID of folder found.
```

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

```
function FSMakeFSSpec  (vRefNum,dirID,fileName,spec) : OSErr;
vRefNum :   integer;      Volume reference number.
dirID :     longint;      Parent directory ID.
fileName :  string;       Full or partial pathname.
Spec :      FSSpec;       Pointer to FSSpec record.
```

- 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 record).

  - Call `SetFPos` to set the file mark to the beginning of the file.

  - Call `FSWrite` to write the buffer to the file:

```
function FSWrite (refNum,count,buffPtr) : OSErr;
refNum :   integer;    File reference number.
Count :    longint;    On input: bytes to write.  On output: bytes written.
BuffPtr :  UNIV Ptr;   Address of buffer containing data to write.
```

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

```
function SetEOF (refNum,logEOF) : OSErr;;
refNum :   integer;    File reference number.
logEOF :   longint;    Logical end-of-file.
```

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

```
function GetVRefNum  (refNum,vRefNum) : OSErr;;
refNum :   integer;    File reference number.
VRefNum :  integer;    Receives volume reference number.
```

  - Call `FlushVol` to flush the volume:

```
function FlushVol (volName,vRefNum) : OSErr;;
volName : Str63;       Pointer to name of mounted volume
vRefNum :  integer;    Volume reference number.
```

    Flushing the volume ensures that both the file's data and the file's catalog entry[2] are updated.

- Call `FSClose` to close the temporary file.

- Call `FSClose` to close the existing file.

- Call `FSpExchangeFiles` to swap[3] the contents of the temporary file and the existing file:

```
function FSpExchangeFiles   (source,dest) : OSErr;
source :  FSSpec;          Source file.
dest :    FSSpec;          Destination file.
```

- Call `FSpDelete` to delete the temporary file:

```
function FSpDelete  (spec) : OSErr;
spec :    FSSpec;          File system specification.
```

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

As a final step, and if the existing file is a newly created file which has been written to for the first time, the function for updating a file should also call an application-defined function which copies the

[2]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.

[3]`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).

missing application name string resource[4] 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:

```
procedure FSpCreateResFile   (spec,creator,fileType,sciptTag);
spec :      FSSpec;       File system specification record.
Creator :   OSType;       File creator.
fileType :  OSType;       File type.
scriptTag : ScriptCode;   Code of script system.
```

- Call `FSpOpenResFile` to open the resource fork:

```
short  FSpOpenResFile  (spec,permission);
spec :        FSSpec;       File system specification record.
Permission :  SignedByte;   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.

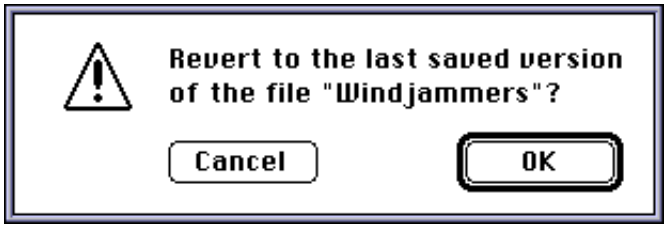

FIG 7 - A REVERT-TO-SAVED DIALOG BOX

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 "dirty" (that is, whether the contents of the window have been changed since the last time it was saved) by checking the `windowDirty` field of the document record.

[4]See Chapter 7 — Finder Interface.

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 record indicates that a file has previously been opened for the document (that is, the file reference number field of the document record 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 record to 0.

- Release memory associated with the storage of the file's data. Then dispose of the document record 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 routines, 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 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.

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.

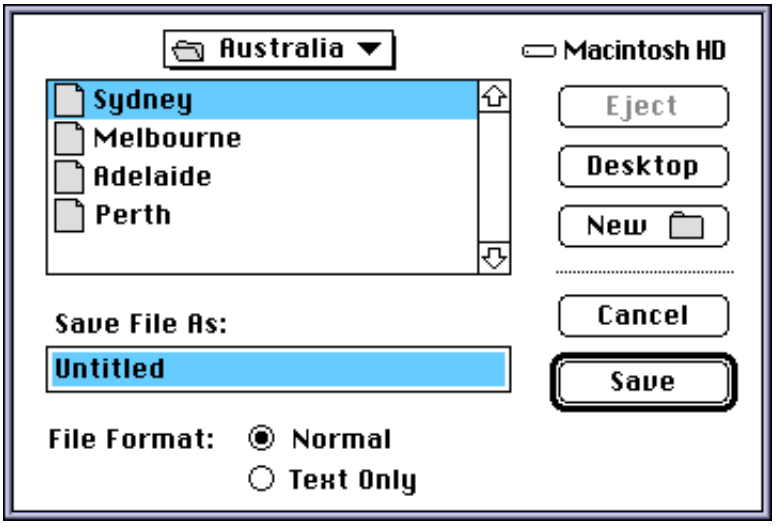

FIG 8 - A CUSTOMIZED SAVE DIALOG BOX

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 button to the standard Open dialog box together with a **Select a Folder:** prompt at the top of the dialog box. By clicking this button, the user would be able to select a highlighted directory rather than open it.

Customising the Standard Dialog Boxes

To customise a dialog box, you should:

- Design your dialog box and create the resources which describe it.
- Write callback routines, if necessary, to process user actions in the dialog box.
- Call the Standard File Package, using `CustomGetFile` and `CustomPutFile`, passing the resource IDs of the customised dialog boxes and pointers to the callback routines:

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

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


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

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

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

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

Main Standard File Package Data Types and Routines

Data Types

```
ConstSFTypeListPtr = ^OSType; { Pointer to an array of OSTypes.}
```

Standard File Reply Record

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

Routines

Saving Files

```
procedure   StandardPutFile(prompt: ConstStr255Param; defaultName: ConstStr255Param;
            VAR reply: StandardFileReply);
procedure   CustomPutFile(prompt: ConstStr255Param; defaultName: ConstStr255Param;
            VAR reply: StandardFileReply; dlgID: integer; where: Point;
            dlgHook: DlgHookYDUPP; filterProc: ModalFilterYDUPP;
            activeList: ActivationOrderListPtr; activate: ActivateYDUPP;
            yourDataPtr: UNIV Ptr);
```

Opening Files

```
procedure   StandardGetFile(fileFilter: FileFilterUPP; numTypes: integer;
            typeList: ConstSFTypeListPtr; VAR reply: StandardFileReply);
procedure   CustomGetFile(fileFilter: FileFilterYDUPP; numTypes: integer;
            typeList: ConstSFTypeListPtr; VAR reply: StandardFileReply; dlgID: integer;
            where: Point; dlgHook: DlgHookYDUPP; filterProc: ModalFilterYDUPP;
            activeList: ActivationOrderListPtr; activate: ActivateYDUPP;
            yourDataPtr: UNIV Ptr);
```

Main File Manager Constants, Data Types and Routines

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 Record

```
FSSpec = record
   vRefNum:     integer;   { Volume reference number.}
   parID:       longint;   { Directory ID of parent directory.}
   name:        Str63;     { Filename or directory name.}
   end;

FSSpecPtr = ^FSSpec;
FSSpecHandle = ^FSSpecPtr;
```

File Information Record

```
FInfo = record
   fdType:      OSType;    { File type.}
   fdCreator:   OSType;    { File's creator.}
   fdFlags:     integer;   { Finder flags (fHasBundle, fInvisible, etc).}
   fdLocation:  Point;     { Position of top-left corner of file's icon.}
   fdFldr:      integer;   { Folder containing file.}
   end;
```

Routines

Reading, Writing and Closing Files

```
function    FSClose(refNum: integer): OSErr;
function    FSRead(refNum: integer; VAR count: longint; buffPtr: UNIV Ptr): OSErr;
function    FSWrite(refNum: integer; VAR count: longint; buffPtr: UNIV Ptr): OSErr;
```

Manipulating the File Mark

```
function    GetFPos(refNum: integer; VAR filePos: longint): OSErr;
function    SetFPos(refNum: integer; posMode: integer; posOff: longint): OSErr;
```

Manipulating the End-Of-File

```
function    GetEOF(refNum: integer; VAR logEOF: longint): OSErr;
function    SetEOF(refNum: integer; logEOF: longint): OSErr;
```

Opening, Creating and Deleting Files

```
function    FSpOpenDF(VAR spec: FSSpec; permission: ByteParameter; VAR refNum: integer): OSErr;
function    FSpOpenRF(VAR spec: FSSpec; permission: ByteParameter; VAR refNum: integer): OSErr;
function    FSpCreate(VAR spec: FSSpec; creator: OSType; fileType: OSType;
            scriptTag: ScriptCode): OSErr;
function    FSpDelete({CONST}VAR spec: FSSpec): OSErr;
```

Exchanging Data in Two Files

```
function    FSpExchangeFiles(VAR source: FSSpec; VAR dest: FSSpec): OSErr;
```

Creating File System Specifications

```
function    FSMakeFSSpec(vRefNum: integer; dirID: longint; fileName: ConstStr255Param;
            VAR spec: FSSpec): OSErr;
```

Updating Volumes

```
function    PBFlushVolSync(paramBlock: ParmBlkPtr): OSErr;
function    PBFlushVolAsync(paramBlock: ParmBlkPtr): OSErr;
```

Obtaining Volume Information

```
function    GetVInfo(drvNum: integer; volName: StringPtr; VAR vRefNum: integer;
            VAR freeBytes: longint): OSErr;
function    GetVRefNum(fileRefNum: integer; VAR vRefNum: integer): OSErr;
```

Relevant Resource Manager Routines

Creating and Opening Resource Files

```
procedure   FSpCreateResFile(VAR spec: FSSpec; creator: OSType; fileType: OSType;
            scriptTag: ScriptCode);
function    FSpOpenResFile(VAR spec: FSSpec; permission: ByteParameter): integer;
```

Relevant Finder Interface Routines

Find a Specified Folder

```
function    FindFolder(vRefNum: integer; folderType: OSType; createFolder: boolean;
            VAR foundVRefNum: integer; VAR foundDirID: longint): OSErr;
```

Demonstration Program

```
{ ############################################################################################
// FilesPascal.p
// ############################################################################################
//
// 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.
//
// •  Receipt of the required Apple events Open Application, Open Documents and Quit
//    Application.
//
// The uncompiled program may be run from within CodeWarrior to demonstrate responses to
// the File menu commands.  The built application, together with supplied TEXT and a 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 keep the code not specifically related to files and file-handling to a minimim, a
// Demonstration menu is included to allow the user to simulate making a window
// 'dirty', that is, modifying the contents of the associated document.  Choosing the
// single menu item in this menu sets the window-dirty flag in the window's document
// record to true and draws a diagonal line across the window, this latter so that the
// user can keep track of which windows are 'dirty'.
//
// 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).
//
// •  Three 'ALRT' resources (purgeable) and associated 'DITL' resources (purgeable).
//    The first alert is used to display error messages.  The second is used to support
//    the Revert to Saved menu item.  The third is used when an attempt is made to close
//    a modified document before that document has been saved.
//
// •  A 'STR ' resource (purgeable) containing the 'missing application name' string
//    resource, which is copied to all document files created by the program.
//
// •  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.
//
// ############################################################################################ }

program FilesPascal(input, output);

{ …… include the following Universal Interfaces }

uses
```

```
54
55    Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
56    Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, Resources, Files, Errors,
57    Folders, Script, StandardFile, AppleEvents, Finder, EPPC, Segload;
58
59  { ……… define the following constants }
60
61  const
62
63  mApple = 128;
64    iAbout = 1;
65  mFile = 129;
66    iNew = 1;
67    iOpen = 2;
68    iClose = 4;
69    iSave = 5;
70    iSaveAs = 6;
71    iRevert = 7;
72    iQuit = 12;
73  mDemonstration = 131;
74    iDirty = 1;
75
76  rNewWindow = 128;
77  rMenubar = 128;
78  rErrorAlert = 128;
79    eInstallHandler = 1000;
80    eMaxWindows = 1001;
81    eFileIsOpen = -49;
82  rRevertAlert = 129;
83  rCloseFileAlert = 130;
84
85  kMaxWindows = 10;
86  kUserCancelled = 1002;
87  kMaxLong = $7FFFFFFF;
88
89  { ……… user defined types }
90
91  type
92
93  DocRecord = record
94    editRec : TEHandle;
95    pictureHdl : PicHandle;
96    fileRefNum : integer;
97    fileFSSpec : FSSpec;
98    windowDirty : boolean;
99    end;
100 DocRecordPointer = ^DocRecord;
101 DocRecordHandle = ^DocRecordPointer;
102
103 { ……… global variables }
104
105 var
106
107 gDone : boolean;
108 gInBackground : boolean;
109 gWindowPtr : WindowPtr;
110 gCurrentNumberOfWindows : longint;
111 gDestRect, gViewRect : Rect;
112 gAppResFileRefNum : longint;
113 menubarHdl : Handle;
114 menuHdl : MenuHandle;
115
116 { ############################################################################ DoInitManagers }
117
118 procedure DoInitManagers;
119
120   begin
121   MaxApplZone;
122   MoreMasters;
123
124   InitGraf(@qd.thePort);
125   InitFonts;
126   InitWindows;
127   InitMenus;
128   TEInit;
129   InitDialogs(nil);
130
```

```
  InitCursor;
  FlushEvents(everyEvent, 0);
  end;
    {of procedure DoInitManagers}

{ ############################################################################ DoCopyResource }

function DoCopyResource(resourceType : ResType; resourceID, sourceFileRefNum,
                        destFileRefNum : integer) : OSErr;

  var
  sourceResourceHdl : Handle;
  sourceResourceName : string;
  ignoredType : ResType;
  ignoredID : integer;

  begin
  UseResFile(sourceFileRefNum);

  sourceResourceHdl := GetResource(resourceType, resourceID);

  if (sourceResourceHdl <> nil) then
    begin
    GetResInfo(sourceResourceHdl, ignoredID, ignoredType, sourceResourceName);
    DetachResource(sourceResourceHdl);
    UseResFile(destFileRefNum);
    AddResource(sourceResourceHdl, resourceType, resourceID, sourceResourceName);
    if (ResError = noErr) then
      UpdateResFile(destFileRefNum);
    end;

  ReleaseResource(sourceResourceHdl);

  DoCopyResource := ResError;
  end;
    {of function DoCopyResource}

{ ################################################################################ DoError }

procedure DoError(errorCode : integer);

  var
  errorString : string;
  ignored : OSErr;

  begin
  if (errorCode = eInstallHandler) then
    ParamText('Error installing Required Apple event handlers', '', '', '')
  else if (errorCode = eMaxWindows) then
    ParamText('Sorry.  No more windows', '', '', '')
  else if (errorCode = eFileIsOpen) then
    ParamText('File is currently open', '', '', '')
  else
    begin
    NumToString(errorCode, errorString);
    ParamText('An error occurred.  The error number is ', errorString, '', '');
    end;

  if (errorCode = memFullErr) then
    begin
    ignored := StopAlert(rErrorAlert, nil);
    ExitToShell;
    end
  else
    ignored := CautionAlert(rErrorAlert, nil);
  end;
    {of procedure DoInitManagers}

{ ################################################################ DoCopyAppNameResource }

function DoCopyAppNameResource(myWindowPtr : WindowPtr) : OSErr;

  var
  docRecHdl : DocRecordHandle;
  fileType : OSType;
  osError : OSErr;
  fileRefNum : integer;
```

```
208
209     begin
210     docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));
211
212     if (docRecHdl^^.editRec <> nil) then
213       fileType := 'TEXT'
214     else if (docRecHdl^^.pictureHdl <> nil) then
215       fileType := 'PICT';
216
217     FSpCreateResFile(docRecHdl^^.fileFSSpec, 'KKJB', fileType, smSystemScript);
218
219     osError := ResError;
220     if (osError = noErr) then
221       fileRefNum := FSpOpenResFile(docRecHdl^^.fileFSSpec, fsRdWrPerm);
222
223     if (fileRefNum > 0) then
224       osError := DoCopyResource('STR ', -16396, gAppResFileRefNum, fileRefNum)
225     else
226       osError := ResError;
227
228     if (osError = noErr) then
229       CloseResFile(fileRefNum);
230
231     osError := ResError;
232     DoCopyAppNameResource := osError;
233     end;
234       {of function DoCopyAppNameResource}
235
236   { ########################################################################## DoWritePictData }
237
238   function  DoWritePictData(myWindowPtr : WindowPtr; tempFileRefNum : integer) : OSErr;
239
240     var
241     docRecHdl : DocRecordHandle;
242     pictureHdl : PicHandle;
243     numberOfBytes, dummyData : longint;
244     volRefNum : integer;
245     osError : OSErr;
246
247     begin
248     docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));
249     pictureHdl := docRecHdl^^.pictureHdl;
250
251     numberOfBytes := 512;
252     dummyData := 0;
253
254     osError := SetFPos(tempFileRefNum, fsFromStart, 0);
255
256     if (osError = noErr) then
257       osError := FSWrite(tempFileRefNum, numberOfBytes, dummyData);
258
259     numberOfBytes := GetHandleSize(Handle(docRecHdl^^.pictureHdl));
260
261     if (osError = noErr) then
262       begin
263       HLock(Handle(docRecHdl^^.pictureHdl));
264       osError := FSWrite(tempFileRefNum, numberOfBytes, docRecHdl^^.pictureHdl^);
265       HUnlock(Handle(docRecHdl^^.pictureHdl));
266       end;
267
268     if (osError = noErr) then
269       osError := SetEOF(tempFileRefNum, 512 + numberOfBytes);
270     if (osError = noErr) then
271       osError := GetVRefNum(tempFileRefNum, volRefNum);
272     if (osError = noErr) then
273       osError := FlushVol(nil, volRefNum);
274
275     if (osError = noErr) then
276       docRecHdl^^.windowDirty := false;
277
278     DoWritePictData := osError;
279     end;
280       {of function DoWritePictData}
281
282   { ########################################################################## DoWriteTextData }
283
284   function DoWriteTextData(myWindowPtr : WindowPtr; tempFileRefNum : integer) : OSErr;
```

```
285
286     var
287     docRecHdl : DocRecordHandle;
288     textEditHdl : TEHandle;
289     editText : Handle;
290     numberOfBytes : longint;
291     volRefNum : integer;
292     osError : OSErr;
293
294     begin
295     docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));
296     textEditHdl := docRecHdl^^.editRec;
297     editText := textEditHdl^^.hText;
298     numberOfBytes := textEditHdl^^.teLength;
299
300     osError := SetFPos(tempFileRefNum, fsFromStart, 0);
301     if (osError = noErr) then
302       osError := FSWrite(tempFileRefNum, numberOfBytes, editText^);
303     if (osError = noErr) then
304       osError := SetEOF(tempFileRefNum, numberOfBytes);
305     if (osError = noErr) then
306       osError := GetVRefNum(tempFileRefNum, volRefNum);
307     if (osError = noErr) then
308       osError := FlushVol(nil, volRefNum);
309
310     if (osError = noErr) then
311       docRecHdl^^.windowDirty := false;
312
313     DoWriteTextData := osError;
314     end;
315       {of function DoWriteTextData}
316
317   { ############################################################################ DoReadPictFile }
318
319   function DoReadPictFile(myWindowPtr : WindowPtr) : OSErr;
320
321     var
322     docRecHdl : DocRecordHandle;
323     fileRefNum : integer;
324     numberOfBytes : longint;
325     osError, ignored : OSErr;
326
327     begin
328     docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));
329     fileRefNum := docRecHdl^^.fileRefNum;
330
331     ignored := GetEOF(fileRefNum, numberOfBytes);
332     ignored := SetFPos(fileRefNum, fsFromStart, 512);
333     numberOfBytes := numberOfBytes - 512;
334
335     docRecHdl^^.pictureHdl := PicHandle(NewHandle(numberOfBytes));
336     if (docRecHdl^^.pictureHdl = nil) then
337       begin
338       DoReadPictFile := MemError;
339       Exit(DoReadPictFile);
340       end;
341
342     osError := FSRead(fileRefNum, numberOfBytes, docRecHdl^^.pictureHdl^);
343     if ((osError = noErr) or (osError = eofErr)) then
344       begin
345       DoReadPictFile := noErr;
346       Exit(DoReadPictFile);
347       end
348     else
349       DoReadPictFile := osError;
350     end;
351       {of function DoReadPictFile}
352
353   { ############################################################################ DoReadTextFile }
354
355   function DoReadTextFile(myWindowPtr : WindowPtr) : OSErr;
356
357     var
358     docRecHdl : DocRecordHandle;
359     fileRefNum : integer;
360     textEditHdl : TEHandle;
361     numberOfBytes : longint;
```

```
362     textBuffer : Handle;
363     osError, ignored : OSErr;
364
365     begin
366     docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));
367     fileRefNum := docRecHdl^^.fileRefNum;
368
369     textEditHdl := docRecHdl^^.editRec;
370     textEditHdl^^.txSize := 10;
371     textEditHdl^^.lineHeight := 15;
372
373     ignored := SetFPos(fileRefNum, fsFromStart, 0);
374     ignored := GetEOF(fileRefNum, numberOfBytes);
375
376     if (numberOfBytes > 32767) then
377       numberOfBytes := 32767;
378
379     textBuffer := NewHandle(Size(numberOfBytes));
380     if (textBuffer = nil) then
381       begin
382       DoReadTextFile := MemError;
383       Exit(DoReadTextFile);
384       end;
385
386     osError := FSRead(fileRefNum, numberOfBytes, textBuffer^);
387     if ((osError = noErr) or (osError = eofErr)) then
388       begin
389       MoveHHi(textBuffer);
390       HLockHi(textBuffer);
391       TESetText(textBuffer^, numberOfBytes, docRecHdl^^.editRec);
392       HUnlock(textBuffer);
393       DisposeHandle(textBuffer);
394       end
395     else
396       begin
397       DoReadTextFile := osError;
398       Exit(DoReadTextFile);
399       end;
400
401     DoReadTextFile := noErr;
402     end;
403       {of function DoReadTextFile}
404
405   { ############################################################################## DoWriteFile }
406
407   function DoWriteFile(myWindowPtr : WindowPtr; newFile : boolean) : OSErr;
408
409     var
410     docRecHdl : DocRecordHandle;
411     fileSpecActual, fileSpecTemp : FSSpec;
412     currentTime : longint;
413     tempFileName : string;
414     tempFileVolNum, tempFileRefNum : integer;
415     tempFileDirID : longint;
416     osError : OSErr;
417
418     begin
419     docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));
420     fileSpecActual := docRecHdl^^.fileFSSpec;
421
422     GetDateTime(currentTime);
423     NumToString(currentTime, tempFileName);
424
425     osError := FindFolder(fileSpecActual.vRefNum, kTemporaryFolderType, kCreateFolder,
426                           tempFileVolNum, tempFileDirID);
427     if (osError = noErr) then
428       osError := FSMakeFSSpec(tempFileVolNum, tempFileDirID, tempFileName, fileSpecTemp);
429     if ((osError = noErr) or (osError = fnfErr)) then
430       osError := FSpCreate(fileSpecTemp, 'trsh', 'trsh', smSystemScript);
431     if (osError = noErr) then
432       osError := FSpOpenDF(fileSpecTemp, fsRdWrPerm, tempFileRefNum);
433     if (osError = noErr) then
434       begin
435       if (docRecHdl^^.editRec <> nil) then
436         osError := DoWriteTextData(myWindowPtr, tempFileRefNum)
437       else if (docRecHdl^^.pictureHdl <> nil) then
438         osError := DoWritePictData(myWindowPtr, tempFileRefNum);
```

```
      end;
    if (osError = noErr) then
      osError := FSClose(tempFileRefNum);
    if (osError = noErr) then
      osError := FSClose(docRecHdl^^.fileRefNum);
    if (osError = noErr) then
      osError := FSpExchangeFiles(fileSpecTemp, fileSpecActual);
    if (osError = noErr) then
      osError := FSpDelete(fileSpecTemp);
    if (osError = noErr) then
      osError := FSpOpenDF(fileSpecActual, fsRdWrPerm, docRecHdl^^.fileRefNum);

    if (osError = noErr) then
      begin
      if (newFile) then
        osError := DoCopyAppNameResource(myWindowPtr);
      end;

    DoWriteFile := osError;
    end;
      {of function DoWriteFile}

{ ############################################################################ DoSaveAsCommand }

function DoSaveAsCommand : OSErr;

    var
    myWindowPtr : WindowPtr;
    docRecHdl : DocRecordHandle;
    fileReply : StandardFileReply;
    fileType : OSType;
    fileRefNum : integer;
    osError : OSErr;

    begin
    myWindowPtr := FrontWindow;
    docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));

    StandardPutFile('Save as:', 'Untitled', fileReply);

    if (fileReply.sfGood) then
      begin
      if not (fileReply.sfReplacing) then
        begin
        if (docRecHdl^^.editRec <> nil) then
          fileType := 'TEXT'
        else if (docRecHdl^^.pictureHdl <> nil) then
          fileType := 'PICT';
        osError := FSpCreate(fileReply.sfFile, 'KKJB', fileType, smSystemScript);
        if (osError <> noErr) then
          begin
          DoSaveAsCommand := osError;
          Exit(DoSaveAsCommand);
          end;
        end;

      docRecHdl^^.fileFSSpec := fileReply.sfFile;

      if (docRecHdl^^.fileRefNum <> 0) then
        begin
        osError := FSClose(docRecHdl^^.fileRefNum);
        docRecHdl^^.fileRefNum := 0;
        end;

      if (osError = noErr) then
        osError := FSpOpenDF(docRecHdl^^.fileFSSpec, fsRdWrPerm, fileRefNum);

      if (osError = noErr) then
        begin
        docRecHdl^^.fileRefNum := fileRefNum;
        SetWTitle(myWindowPtr, fileReply.sfFile.name);
        osError := DoWriteFile(myWindowPtr, true);
        end;
      end;

    DoSaveAsCommand := osError;
    end;
```

```
516       {of function DoSaveAsCommand}
517
518   { ############################################################################## DoSaveCommand }
519
520   function DoSaveCommand : OSErr;
521
522     var
523     myWindowPtr : WindowPtr;
524     docRecHdl : DocRecordHandle;
525     osError : OSErr;
526
527     begin
528     osError := 0;
529     myWindowPtr := FrontWindow;
530     docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));
531
532     if (docRecHdl^^.fileRefNum <> 0) then
533       osError := DoWriteFile(myWindowPtr, false)
534     else
535       osError := DoSaveAsCommand;
536
537     DoSaveCommand := osError;
538     end;
539       {of function DoSaveCommand}
540
541   { ############################################################################## DoCloseFile }
542
543   function DoCloseFile(myWindowPtr : WindowPtr; docRecHdl : DocRecordHandle) : OSErr;
544
545     var
546     fileName : string;
547     itemHit : integer;
548     osError : OSErr;
549
550     begin
551     if (docRecHdl^^.windowDirty) then
552       begin
553       GetWTitle(myWindowPtr, fileName);
554       ParamText(fileName, '', '', '');
555
556       itemHit := CautionAlert(rCloseFileAlert, nil);
557       if (itemHit = 2) then
558         begin
559         DoCloseFile := kUserCancelled;
560         Exit(DoCloseFile);
561         end
562       else if (itemHit = 1) then
563         begin
564         osError := DoSaveCommand;
565         if (osError <> noErr) then
566           begin
567           DoCloseFile := osError;
568           Exit(DoCloseFile);
569           end;
570         end;
571       end;
572
573     if (docRecHdl^^.fileRefNum <> 0) then
574       begin
575       osError := FSClose(docRecHdl^^.fileRefNum);
576       if (osError <> noErr) then
577         begin
578         osError := FlushVol(nil, docRecHdl^^.fileFSSpec.vRefNum);
579         docRecHdl^^.fileRefNum := 0;
580         end;
581       end;
582
583     if (docRecHdl^^.editRec <> nil) then
584       TEDispose(docRecHdl^^.editRec);
585     if (docRecHdl^^.pictureHdl <> nil) then
586       KillPicture(docRecHdl^^.pictureHdl);
587
588     DisposeHandle(Handle(docRecHdl));
589
590     DoCloseFile := osError;
591     end;
592       {of function DoCloseFile}
```

```
593
594 { ############################################################################ DoNewDocWindow }
595
596 function DoNewDocWindow(showWindow : boolean; documentType : OSType) : OSErr;
597
598   var
599   docRecHdl : DocRecordHandle;
600
601   begin
602   if (gCurrentNumberOfWindows = kMaxWindows) then
603     begin
604     DoNewDocWindow := eMaxWindows;
605     Exit(DoNewDocWindow);
606     end;
607
608   gWindowPtr := GetNewWindow(rNewWindow, nil, WindowPtr(-1));
609   if (gWindowPtr = nil) then
610     begin
611     DoNewDocWindow := MemError;
612     Exit(DoNewDocWindow);
613     end;
614
615   SetPort(gWindowPtr);
616
617   docRecHdl := DocRecordHandle(NewHandle(sizeof(DocRecord)));
618   if (docRecHdl = nil) then
619     begin
620     DisposeWindow(gWindowPtr);
621     gCurrentNumberOfWindows := gCurrentNumberOfWindows -1;
622     DoNewDocWindow := MemError;
623     Exit(DoNewDocWindow);
624     end;
625
626   SetWRefCon(gWindowPtr, longint(docRecHdl));
627
628   docRecHdl^^.editRec := nil;
629   docRecHdl^^.pictureHdl := nil;
630   docRecHdl^^.fileRefNum := 0;
631   docRecHdl^^.windowDirty := false;
632
633   if (documentType = 'TEXT') then
634     begin
635     gDestRect := gWindowPtr^.portRect;
636     InsetRect(gDestRect, 6, 6);
637     gViewRect := gDestRect;
638
639     MoveHHi(Handle(docRecHdl));
640     HLock(Handle(docRecHdl));
641
642     docRecHdl^^.editRec := TENew(gDestRect, gViewRect);
643     if (docRecHdl^^.editRec = nil) then
644       begin
645       DisposeWindow(gWindowPtr);
646       gCurrentNumberOfWindows := gCurrentNumberOfWindows -1;
647       DisposeHandle(Handle(docRecHdl));
648       DoNewDocWindow := MemError;
649       Exit(DoNewDocWindow);
650       end;
651
652     HUnlock(Handle(docRecHdl));
653     end;
654
655   if (showWindow) then
656     ShowWindow(gWindowPtr);
657
658   gCurrentNumberOfWindows := gCurrentNumberOfWindows + 1;
659
660   DoNewDocWindow := noErr;
661   end;
662     {of function DoNewDocWindow}
663
664 { ############################################################################ DoOpenFile }
665
666 function DoOpenFile(fileSpec : FSSpec; documentType : OSType) : OSErr;
667
668   var
669   osError : OSErr;
```

```
670     fileRefNum : integer;
671     docRecHdl : DocRecordHandle;
672
673     begin
674     osError := DoNewDocWindow(false, documentType);
675     if (osError <> noErr) then
676       begin
677       DoOpenFile := osError;
678       Exit(DoOpenFile);
679       end;
680
681     SetWTitle(gWindowPtr, fileSpec.name);
682
683     osError := FSpOpenDF(fileSpec, fsRdWrPerm, fileRefNum);
684     if (osError <> noErr) then
685       begin
686       DisposeWindow(gWindowPtr);
687       gCurrentNumberOfWindows := gCurrentNumberOfWindows - 1;
688       DoOpenFile := osError;
689       Exit(DoOpenFile);
690       end;
691
692     docRecHdl := DocRecordHandle(GetWRefCon(gWindowPtr));
693     docRecHdl^^.fileRefNum := fileRefNum;
694     docRecHdl^^.fileFSSpec := fileSpec;
695
696     if (documentType = 'TEXT') then
697       begin
698       osError := DoReadTextFile(gWindowPtr);
699       if (osError <> noErr) then
700         begin
701         DoOpenFile := osError;
702         Exit(DoOpenFile);
703         end;
704       end
705     else if (documentType = 'PICT') then
706       begin
707       osError := DoReadPictFile(gWindowPtr);
708       if (osError <> noErr) then
709         begin
710         DoOpenFile := osError;
711         Exit(DoOpenFile);
712         end;
713       end;
714
715     ShowWindow(gWindowPtr);
716
717     DoOpenFile := noErr;
718     end;
719       {of function DoOpenFile}
720
721   { ############################################################################## DoCloseCommand }
722
723   function DoCloseCommand : OSErr;
724
725     var
726     myWindowPtr : WindowPtr;
727     windowKind : integer;
728     docRecHdl : DocRecordHandle;
729     osError : OSErr;
730
731     begin
732     osError := 0;
733     myWindowPtr := FrontWindow;
734     windowKind := WindowPeek(myWindowPtr)^.windowKind;
735
736     case (windowKind) of
737
738       userKind:
739         begin
740         docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));
741         osError := DoCloseFile(myWindowPtr, docRecHdl);
742         if (osError = kUserCancelled) then
743           begin
744           DoCloseCommand := kUserCancelled;
745           Exit(DoCloseCommand);
746           end
```

```
747         else if (osError = noErr) then
748           begin
749           DisposeWindow(myWindowPtr);
750           gCurrentNumberOfWindows := gCurrentNumberOfWindows - 1;
751           end;
752         end;
753 
754       dialogKind:
755         begin
756         { Hide or close modeless dialog, as required.}
757         end;
758       end;
759         {of case statement}
760 
761     DoCloseCommand := osError;
762     end;
763       {of function DoCloseCommand}
764 
765   { ############################################################################ DoQuitCommand }
766 
767   function DoQuitCommand : OSErr;
768 
769     var
770     osError : OSErr;
771 
772     begin
773     osError := 0;
774     while (FrontWindow <> nil) do
775       begin
776       osError := DoCloseCommand;
777       if (osError <> noErr) then
778         begin
779         DoQuitCommand := osError;
780         Exit(DoQuitCommand);
781         end;
782       end;
783 
784     DoQuitCommand := osError;
785     end;
786       {of function DoQuitCommand}
787 
788   { ############################################################################ DoRevertCommand }
789 
790   function  DoRevertCommand : OSErr;
791 
792     var
793     myWindowPtr : WindowPtr;
794     docRecHdl : DocRecordHandle;
795     fileRefNum : integer;
796     fileName : string;
797     itemHit : integer;
798     osError : OSErr;
799 
800     begin
801     osError := 0;
802     myWindowPtr := FrontWindow;
803     docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));
804     fileRefNum := docRecHdl^^.fileRefNum;
805 
806     SetPort(myWindowPtr);
807 
808     GetWTitle(myWindowPtr, fileName);
809     ParamText(fileName, '', '', '');
810 
811     SetPort(myWindowPtr);
812 
813     itemHit := CautionAlert(rRevertAlert, nil);
814 
815     if (itemHit = 1) then
816       begin
817       EraseRect(myWindowPtr^.portRect);
818       if (docRecHdl^^.editRec <> nil) then
819         osError := DoReadTextFile(myWindowPtr)
820       else if (docRecHdl^^.pictureHdl <> nil) then
821         begin
822         KillPicture(docRecHdl^^.pictureHdl);
823         docRecHdl^^.pictureHdl := nil;
```

```
824         osError := DoReadPictFile(myWindowPtr);
825         end;
826
827       docRecHdl^^.windowDirty := false;
828
829       InvalRect(myWindowPtr^.portRect);
830       end;
831
832     DoRevertCommand := osError;
833     end;
834       {of function DoRevertCommand}
835
836   { ############################################################################ DoOpenCommand }
837
838   function DoOpenCommand : OSErr;
839
840     var
841     fileTypes : SFTypeList;
842     fileReply : StandardFileReply;
843     documentType : OSType;
844      osError : OSErr;
845
846     begin
847     osError := 0;
848     fileTypes[0] := 'TEXT';
849     fileTypes[1] := 'PICT';
850
851     StandardGetFile(nil, 2, ConstSFTypeListPtr(@fileTypes), fileReply);
852
853     documentType := fileReply.sfType;
854
855     if (fileReply.sfGood) then
856       osError := DoOpenFile(fileReply.sfFile, documentType);
857
858     DoOpenCommand := osError;
859     end;
860       {of function DoOpenCommand}
861
862   { ############################################################################ DoNewCommand }
863
864   function DoNewCommand : OSErr;
865
866     var
867     osError : OSErr;
868     documentType : OSType;
869
870     begin
871     documentType := 'TEXT';
872     osError := DoNewDocWindow(true, documentType);
873     DoNewCommand := osError;
874     end;
875       {of function DoNewCommand}
876
877   { ##################################################################### HasGotRequiredParams }
878
879   function HasGotRequiredParams(appEvent : AppleEvent) : OSErr;
880
881     var
882     returnedType : DescType;
883     actualSize : Size;
884     osError : OSErr;
885
886     begin
887     osError := AEGetAttributePtr(appEvent, keyMissedKeywordAttr, typeWildCard, returnedType,
888                           nil, 0, actualSize);
889     if (osError = errAEDescNotFound) then
890       HasGotRequiredParams := noErr
891     else if (osError = noErr) then
892       HasGotRequiredParams := errAEParamMissed;
893     end;
894       {of function HasGotRequiredParams}
895
896   { ############################################################################ DoOpenDocsEvent }
897
898   function DoOpenDocsEvent(var appEvent, reply : AppleEvent; handlerRefcon : longint) : OSErr;
899
900     var
```

```
  fileSpec : FSSpec;
  docList : AEDescList;
  osError, ignoreErr : OSErr;
  index, numberOfItems : longint;
  actualSize : Size;
  keyWord : AEKeyword;
  returnedType : DescType;
  fileInfo : FInfo;

  begin
  osError := AEGetParamDesc(appEvent, keyDirectObject, typeAEList, docList);

  if (osError = noErr) then
    begin
    osError := HasGotRequiredParams(appEvent);
    if (osError = noErr) then
      begin
      ignoreErr := AECountItems(docList, numberOfItems);
      if (osError = noErr) then
        begin
        for index := 1 to numberOfItems do
          begin
          osError := AEGetNthPtr(docList, index, typeFSS, keyWord, returnedType,
                     @fileSpec, sizeof(fileSpec), actualSize);
          if (osError = noErr) then
            begin
            osError := FSpGetFInfo(fileSpec, fileInfo);
            if (osError = noErr) then
              begin
              osError := DoOpenFile(fileSpec, fileInfo.fdType);
              if (osError <> noErr) then
                DoError(osError);
              end;
            end
          else
            DoError(osError);
          end;
        end;
      end
    else
      DoError(osError);

    ignoreErr := AEDisposeDesc(docList);
    end

  else
    DoError(osError);

  DoOpenDocsEvent := osError;
  end;
    {of function DoOpenDocsEvent}

{ ############################################################################ DoOpenAppEvent }

function DoOpenAppEvent(var appEvent, reply : AppleEvent; handlerRefCon : longint)
                   : OSErr;

  var
  osError : OSErr;

  begin
  osError := HasGotRequiredParams(appEvent);
  if (osError = noErr) then
    osError := DoNewCommand;

  DoOpenAppEvent := osError;
  end;
    {of function DoOpenAppEvent}

{ ############################################################################ DoMakeWindowDirty }

procedure DoMakeWindowDirty;

  var
  myWindowPtr : WindowPtr;
  docRecHdl : DocRecordHandle;
```

```
  begin
  myWindowPtr := FrontWindow;
  docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));

  SetPort(myWindowPtr);
  MoveTo(myWindowPtr^.portRect.left, myWindowPtr^.portRect.top);
  LineTo(myWindowPtr^.portRect.right, myWindowPtr^.portRect.bottom);

  docRecHdl^^.windowDirty := true;
  end;
    {of procedure DoMakeWindowDirty}

{ ############################################################################ DoAdjustMenus }

procedure DoAdjustMenus;

  var
  menuHdl : MenuHandle;
  myWindowPtr : WindowPtr;
  docRecHdl : DocRecordHandle;

  begin
  myWindowPtr := FrontWindow;
  docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));

  menuHdl := GetMenuHandle(mFile);

  if (gCurrentNumberOfWindows > 0) then
    begin
    menuHdl := GetMenuHandle(mFile);
    EnableItem(menuHdl, iClose);
    EnableItem(menuHdl, iSave);
    EnableItem(menuHdl, iSaveAs);
    EnableItem(menuHdl, iRevert);

    {Note the use of the short-circuit boolean 'and' operator. Something to be aware
    of when porting C to Pascal.}
    if ((docRecHdl^^.pictureHdl = nil) & (docRecHdl^^.editRec^^.teLength = 0))
      then DisableItem(menuHdl, iRevert);

    menuHdl := GetMenuHandle(mDemonstration);
    if not (docRecHdl^^.windowDirty) then
      EnableItem(menuHdl, iDirty)
    else
      DisableItem(menuHdl, iDirty);
    end
  else
    begin
    menuHdl := GetMenuHandle(mFile);
    DisableItem(menuHdl, iClose);
    DisableItem(menuHdl, iSave);
    DisableItem(menuHdl, iSaveAs);
    DisableItem(menuHdl, iRevert);
    menuHdl := GetMenuHandle(mDemonstration);
    DisableItem(menuHdl, iDirty);
    end;

  DrawMenuBar;
  end;
    {of procedure DoAdjustMenus}

{ ############################################################################## DoFileMenu }

procedure DoFileMenu(menuItem : longint);

  var
  osError : OSErr;

  begin
  case (menuItem) of

    iNew:
      begin
      osError := DoNewCommand;
      if (osError <> noErr) then
        DoError(osError);
      end;
```

```
1055
1056        iOpen:
1057          begin
1058          osError := DoOpenCommand;
1059          if (osError <> noErr) then
1060            DoError(osError);
1061          end;
1062
1063        iClose:
1064          begin
1065          osError := DoCloseCommand;
1066          if ((osError <> noErr) and not (osError = kUserCancelled)) then
1067            DoError(osError);
1068          end;
1069
1070        iSave:
1071          begin
1072          osError := DoSaveCommand;
1073          if (osError <> noErr) then
1074            DoError(osError);
1075          end;
1076
1077        iSaveAs:
1078          begin
1079          osError := DoSaveAsCommand;
1080          if (osError <> noErr) then
1081            DoError(osError);
1082          end;
1083
1084        iRevert:
1085          begin
1086          osError := DoRevertCommand;
1087          if (osError <> noErr) then
1088            DoError(osError);
1089          end;
1090
1091        iQuit:
1092          begin
1093          osError := DoQuitCommand;
1094          if ((osError <> noErr) and not (osError = kUserCancelled)) then
1095            DoError(osError);
1096          if (osError <> kUserCancelled) then
1097            gDone := true;
1098          end;
1099        end;
1100          {of case statement}
1101    end;
1102      {of procedure DoInitManagers}
1103
1104  { ############################################################################ DoMenuChoice }
1105
1106  procedure DoMenuChoice(menuChoice : longint);
1107
1108    var
1109    menuID, menuItem : longint;
1110    itemName : string;
1111    daDriverRefNum : longint;
1112
1113    begin
1114    menuID := HiWord(menuChoice);
1115    menuItem := LoWord(menuChoice);
1116
1117    if (menuID = 0) then
1118      Exit(DoMenuChoice);
1119
1120    case (menuID) of
1121
1122       mApple:
1123        begin
1124        GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
1125        daDriverRefNum := OpenDeskAcc(itemName);
1126        end;
1127
1128      mFile:
1129        begin
1130        DoFileMenu(menuItem);
1131        end;
```

```
1132
1133      mDemonstration:
1134        begin
1135        DoMakeWindowDirty;
1136        end;
1137      end;
1138        {of case statement}
1139
1140    HiliteMenu(0);
1141    end;
1142      {of procedure DoMenuChoice}
1143
1144  { ############################################################################## DoUpdate }
1145
1146  procedure DoUpdate(eventRec : EventRecord);
1147
1148    var
1149    myWindowPtr : WindowPtr;
1150    docRecHdl : DocRecordHandle;
1151    destRect : Rect;
1152
1153    begin
1154    myWindowPtr := WindowPtr(eventRec.message);
1155    docRecHdl := DocRecordHandle(GetWRefCon(myWindowPtr));
1156
1157    SetPort(myWindowPtr);
1158
1159    BeginUpdate(myWindowPtr);
1160
1161    if (docRecHdl^^.pictureHdl <> nil) then
1162      begin
1163      destRect := docRecHdl^^.pictureHdl^^.picFrame;
1164      HLock(Handle(docRecHdl^^.pictureHdl));
1165      DrawPicture(docRecHdl^^.pictureHdl, destRect);
1166      HUnlock(Handle(docRecHdl^^.pictureHdl));
1167      end
1168    else if (docRecHdl^^.editRec <> nil) then
1169      begin
1170      HLock(Handle(docRecHdl^^.editRec));
1171      TEUpdate(gDestRect, docRecHdl^^.editRec);
1172      HUnlock(Handle(docRecHdl^^.editRec));
1173      end;
1174
1175    if (docRecHdl^^.windowDirty) then
1176      begin
1177      MoveTo(myWindowPtr^.portRect.left, myWindowPtr^.portRect.top);
1178      LineTo(myWindowPtr^.portRect.right, myWindowPtr^.portRect.bottom);
1179      end;
1180
1181    EndUpdate(myWindowPtr);
1182    end;
1183      {of procedure DoUpdate}
1184
1185  { ########################################################################## DoMouseDown }
1186
1187  procedure DoMouseDown(eventRec : EventRecord);
1188
1189    var
1190    myWindowPtr : WindowPtr;
1191    partCode : longint;
1192    ignored : OSErr;
1193
1194    begin
1195    partCode := FindWindow(eventRec.where, myWindowPtr);
1196
1197    case (partCode) of
1198
1199      inMenuBar:
1200        begin
1201        DoAdjustMenus;
1202        DoMenuChoice(MenuSelect(eventRec.where));
1203        end;
1204
1205      inSysWindow:
1206        begin
1207        SystemClick(eventRec, myWindowPtr);
1208        end;
```

```
1209
1210      inContent:
1211        begin
1212        if (myWindowPtr <> FrontWindow) then
1213          SelectWindow(myWindowPtr);
1214        end;
1215
1216      inDrag:
1217        begin
1218        DragWindow(myWindowPtr, eventRec.where, qd.screenBits.bounds);
1219        end;
1220
1221      inGoAway:
1222        begin
1223        if (TrackGoAway(myWindowPtr, eventRec.where)) then
1224          ignored := DoCloseCommand;
1225        end;
1226      end;
1227        {of case statement}
1228    end;
1229      {of procedure DoMouseDown}
1230
1231  { ############################################################################## DoEvents }
1232
1233  procedure DoEvents(eventRec : EventRecord);
1234
1235    var
1236    charCode : char;
1237    ignored : OSErr;
1238
1239    begin
1240    case (eventRec.what) of
1241
1242      kHighLevelEvent:
1243        begin
1244        ignored := AEProcessAppleEvent(eventRec);
1245        end;
1246
1247      mouseDown:
1248        begin
1249        DoMouseDown(eventRec);
1250        end;
1251
1252      keyDown, autoKey:
1253        begin
1254        charCode := chr(BAnd(eventRec.message, charCodeMask));
1255        if (BAnd(eventRec.modifiers, cmdKey) <> 0) then
1256          begin
1257          DoAdjustMenus;
1258          DoMenuChoice(MenuKey(charCode));
1259          end;
1260        end;
1261
1262      updateEvt:
1263        begin
1264        DoUpdate(eventRec);
1265        end;
1266
1267      osEvt:
1268        begin
1269        case (BAnd(BSR(eventRec.message, 24), $000000FF)) of
1270
1271          suspendResumeMessage:
1272            begin
1273            gInBackground := (BAnd(eventRec.message, resumeFlag) = 0);
1274            end;
1275          end;
1276            {of inner case statement}
1277
1278        HiliteMenu(0);
1279        end;
1280      end;
1281        {of outer case statement}
1282    end;
1283      {of procedure DoEvents}
1284
1285  { ############################################################################## EventLoop }
```

```
1286
1287  procedure EventLoop;
1288
1289    var
1290    eventRec : EventRecord;
1291
1292    begin
1293    gDone := false;
1294
1295    while not (gDone) do
1296      begin
1297      if (WaitNextEvent(everyEvent, eventRec, kMaxLong, nil)) then
1298        DoEvents(eventRec);
1299      end;
1300
1301    end;
1302      {of procedure EventLoop}
1303
1304  { ############################################################################ DoQuitAppEvent }
1305
1306  function DoQuitAppEvent(var appEvent, reply : AppleEvent; handlerRefcon : longint)
1307                  : OSErr;
1308
1309    var
1310    osError : OSErr;
1311    begin
1312    osError := HasGotRequiredParams(appEvent);
1313    if (osError = noErr) then
1314      begin
1315      while (FrontWindow <> nil) do
1316        begin
1317        osError := DoCloseCommand;
1318        if ((osError <> noErr) and (osError <> kUserCancelled)) then
1319          DoError(osError);
1320        if (osError = kUserCancelled) then
1321          EventLoop;
1322          gDone := true;
1323          end;
1324      end;
1325    DoQuitAppEvent := osError;
1326    end;
1327      {of function DoQuitAppEvent}
1328
1329  { ################################################################### DoInstallAEHandlers }
1330
1331  procedure DoInstallAEHandlers;
1332
1333    var
1334    err : OSErr;
1335
1336    begin
1337    err := AEInstallEventHandler(kCoreEventClass, kAEOpenApplication,
1338                    AEEventHandlerUPP(@DoOpenAppEvent), 0, false);
1339
1340    if (err <> noErr) then
1341      DoError(eInstallHandler);
1342
1343    err := AEInstallEventHandler(kCoreEventClass, kAEOpenDocuments,
1344                    AEEventHandlerUPP(@DoOpenDocsEvent), 0, false);
1345
1346    if (err <> noErr) then
1347      DoError(eInstallHandler);
1348
1349    err := AEInstallEventHandler(kCoreEventClass, kAEQuitApplication,
1350                    AEEventHandlerUPP(@DoQuitAppEvent), 0, false);
1351
1352    if (err <> noErr) then
1353      DoError(eInstallHandler);
1354    end;
1355      {of procedure DoInstallAEHandlers}
1356
1357  { ################################################################## start of main program }
1358
1359  begin
1360
1361    { ..... initialise managers }
1362
```

```
1363    DoInitManagers;
1364    gCurrentNumberOfWindows := 0;
1365
1366    { ……… set application's resource fork as current resource file }
1367
1368    gAppResFileRefNum := CurResFile;
1369
1370    { ……… set up menu bar and menus }
1371
1372    menubarHdl := GetNewMBar(rMenubar);
1373    if (menubarHdl = nil) then
1374      DoError(MemError);
1375    SetMenuBar(menubarHdl);
1376    DrawMenuBar;
1377
1378    menuHdl := GetMenuHandle(mApple);
1379    if (menuHdl = nil) then
1380      DoError(MemError)
1381    else
1382      AppendResMenu(menuHdl, 'DRVR');
1383
1384    { ……… install required Apple event handlers }
1385
1386    DoInstallAEHandlers;
1387
1388    { ……… enter EventLoop }
1389
1390    EventLoop;
1391
1392  end.
1393
1394  { ########################################################################################## }
```

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 "dirty" by choosing the Demonstration menu item, reverting to the saved versions of files associated with "dirty" windows, choosing Quit when "dirty" and non-"dirty" windows are open, and so on.

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

To prove that the missing application name string resource is correctly copied to all new files created by the application, SimpleText, TeachText and the Files application should be removed from the hard disk and any mounted floppy disk. Double clicking a document icon should then invoke a "missing application" alert box with the application's name inserted in the advisory text.

The constant declaration block

Lines 63-74 establish constants relating to menu IDs and menu item numbers. Lines 76-83 establish constants relating to window, menu bar and alert resources. Lines 79-81 are constants for some specific error conditions. The constant at Line 85 is used to limit the number of windows the user can open. The constant at Line 86 is used when the user clicks the Cancel button of a particular alert box. Line 87 defines kMaxLong as the maximum possible long value.

The type declaration block

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

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

The variable declaration block

gDone controls termination of the main loop and thus of the program. gInBackground relates to foreground/background switching. 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 records associated with 'TEXT' files. gAppResFileRefNum will be assigned the file reference number of the application's resource fork.

The function DoCopyResource

DoCopyResource copies specified resources between specified files. In this program, it is called only by DoCopyAppNameResource.

Line 148 sets the application's resource fork as the current resource file. Line 150 reads the specified resource into memory.

Line 154, 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.)

Line 155 removes the resource's handle from the resource map without removing the resource from memory, and converts the resource handle into a generic handle. Line 156 makes the new file's resource fork the current resource file. Line 157 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. Line 159 then writes the resource map and data to disk.

The procedure DoError

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

The procedure 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.

Line 210 retrieves the handle to the file's document record. Lines 212-215 establish the file type involved. Line 217 creates the resource fork in the new file and Line 221 opens the resource fork. Line 224 then calls the application-defined function for copying specified resources between specified files. 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.

Line 229 closes the resource fork of the new file.

The function DoWritePictData

DoWritePictData is called by DoWriteFile to write picture data to the specified file.

Lines 248-249 retrieve the handle to the relevant Picture record from the document record. Line 254 sets the file mark to the start of the file. Line 257 writes zeros in the first 512 bytes (the size of a 'PICT' file's header). Line 259 gets the size of the Picture record and Line 264 writes the bytes in the Picture record to the file. Line 269 adjusts the file's size and Lines 271 and 273 store to disk all unwritten data currently in the volume buffer.

Line 276 sets the windowDirty field of the document record to indicate that the document data on disk equates to the document data in memory

The function DoWriteTextData

DoWriteTextData is called by DoWriteFile to write text data to the specified file.

Lines 295-297 retrieve the handle to the TextEdit edit record from the document record. The number of bytes of text is then retrieved from the teLength field of the text edit record (Line 298).

Line 300 sets the file mark to the beginning of the file. Line 302 writes the specified number of bytes to the file. Line 304 adjusts the file's size. Lines 306 and 308 store to disk all unwritten data currently in the volume buffer.

Line 311 sets the windowDirty field of the document record to indicate that the document data on disk equates to the document data in memory

The function DoReadPictFile

DoReadPictFile is called by DoOpenFile and DoRevertCommand to read in data from an open file of type 'PICT'.

Lines 328-329 retrieve the file reference number from the document record. Line 331 gets the number of bytes in the file. Line 332 sets the file mark 512 bytes (the size of a 'PICT' file's header) past the beginning of the file and Line 333 subtracts the header size from the total size of the file. Line 335 allocates memory for the Picture record and Line 342 reads in the file's data.

The function DoReadTextFile

DoReadTextFile is called by DoOpenFile and DoRevertCommand to read in data from an open file of type 'TEXT'.

Lines 366-367 retrieve the file reference number from the document record. Line 369 retrieves the handle to the TextEdit edit record from the document record and Lines 370-371 modify the text size and line height fields of the edit record.

Line 373 sets the file mark to the beginning of the file. Line 374 gets the number of bytes in the file. If the number of bytes exceeds that which can be stored in a TextEdit edit record (32,767), Lines 376-377 restrict the number of bytes which will be read from the file to 32,767.

Line 379 allocates a buffer equal to the size of the file (or 32,767 bytes if Line 377 executed). Line 386 reads the data from the file into the buffer. Lines 389-390 move the buffer high in the heap and lock it preparatory to the call to TESetText at Line 391. TESetText copies the text in the buffer into the existing hText handle of the TextEdit edit record. The buffer is then unlocked and disposed of (Lines 392-393).

(Note: TextEdit is addressed in detail at Chapter 17 - Text and TextEdit.)

The procedure 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.

Line 419 retrieves the handle to the document record and Line 420 retrieves the file system specification from the document record. Lines 422-423 create a temporary file name which is bound to be unique. Line 425 finds the temporary folder on the file's volume, or creates a temporary folder if necessary. Line 428 makes a file system specification record for the temporary file, using the volume reference number and parent directory ID returned by the FindFolder call at Line 425. Line 430 creates the temporary file in that directory on that volume, and Line 432 opens the file's data fork.

Lines 434-439 call the appropriate application-defined function to write the document's data to the temporary file.

Lines 441 and 443 close both the temporary and "actual" files prior to the call to FSpExchangeFiles (Line 445), which swaps the files' data by changing the information in the volume's catalog. The temporary file is then deleted (Line 447) and the data fork of the "actual" file is re-opened (Line 449).

If the file is a newly created file (Line 453), an application-defined function is called (Line 454) 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.

The function 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.

Line 474 gets the WindowPtr for the front window and Line 475 retrieves the handle to that window's document record.

Line 477 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 record "filled-in" by StandardPutFile indicates that an existing file is not being replaced, Lines 483-487 retrieve the file type from the document record for the front window and create a new file of that type, specifying the application's signature as the creator.

Line 495 assigns the file system specification record returned in the sfFile field of the StandardFileReply record to the fileFSSpec field of the document record.

If a file currently exists for the document (Line 497), that file is closed (Lines 499-500).

The data fork of the newly created file is then opened by a call to FSpOpenDF (Line 504), the fileRefNum field of the document record is assigned the file reference number returned by FSpOpenDF (Line 508), the window's title is set to the new file's name (Line 509), and the application-defined function DoWriteFile is called to write the document to the new file (Line 510).

The function DoSaveCommand

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

Line 529 gets the WindowPtr for the front window and Line 530 retrieves the handle to that window's document record. If a file currently exists for the document in this window (Line 532), the application-defined function DoWriteFile is called (Line 533), otherwise the application-defined function DoSaveAsCommand is called (Line 535).

The function DoCloseFile

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

If the window is dirty (Line 551), a caution alert is presented (Line 556) asking the user whether the document should be saved. (Lines 553-554 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 simply returns (Lines 559-560). If the user clicks YES, the application-defined procedure DoSaveCommand is called to save the file (Lines 563-570).

If the user clicks YES or NO:

- If the document has a file (Line 573), Line 575 closes the file, and Line 578 stores to disk all unwritten data currently in the volume buffer.

- If the document is a text document, the text edit record is disposed of (Lines 583-584). If it is a picture document, the Picture record is disposed of (Lines 585-586). Finally, the document record is disposed of (Line 588).

The function DoNewDocWindow

DoNewDocWindow is called by DoNewCommand, DoOpenFile and the Open Application event handler. It creates a new window and associated document record.

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 (Lines 604-605).

Line 608 opens a new window and Line 615 sets the window's graphics port as the current port for drawing.

Line 617 allocates memory for the window's document record. If this call is not successful, the window is disposed of (Line 620) and the function returns with the error code returned by MemError (Line 622).

Line 626 assigns the handle to the document record to the window record's refCon field. Lines 628-631 initialise fields of the document record.

If the document type is 'TEXT' (Line 633), Lines 634-653 create a TextEdit edit record and assign a handle to that record to the editRec field of the document record. (Note that the processes here are not explained in detail because TextEdit and edit records 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 record. TextEdit is addressed in detail at Chapter 17 — Text and TextEdit.)

If the Boolean value passed to DoNewDocWindow was set to true (Line 655), Lines 656 make the window visible, otherwise the window is left invisible. Line 658 increments the global variable which keeps track of the number of open windows.

The function DoOpenFile

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

Line 674 calls DoNewDocWindow to open a new window and create an associated document record. Line 681 sets the window's title. Line 683 opens the file's data fork. If this call is not successful, the window is disposed of and the function returns. Lines 693-694 assign the file reference number and file system specification record to the relevant fields of the document record.

Lines 696-713 call 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, Line 715 makes the window visible.

The function 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.

Line 733 gets the WindowPtr for the front window and Line 734 establishes whether the front window is a document window or a modeless dialog box.

If the front window is a document window (Line 738), the handle to the window's document record is retrieved from the window record's refCon field (Line 740). The WindowPtr and this handle are then passed to the application-defined function DoCloseFile at Line 741. If the window is "dirty", 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 (Line 742), in which case DoCloseCommand simply returns. 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 (Line 749), and the global variable which keeps track of the number of open windows is decremented (Line 750).

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 Line 756.

The function DoQuitCommand

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

The while loop initiated at Line 774 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.

The function DoRevertCommand

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

Line 802 gets the WindowPtr for the front window and Line 803 retrieves the handle to that window's document record. Line 804 retrieves the file reference number from the document record.

Line 808 retrieves the window's title (that is, the filename) for insertion by Line 809 into the text of the alert box invoked at Line 813. (The alert box asks the user to confirm, or otherwise, the reversion to the last saved version.)

If the user clicks the OK button (Line 815), the window is erased (Line 817) and the appropriate application-defined routine (DoReadTextFile or DoReadPictFile) is called depending on whether the file type is 'TEXT' or 'PICT' (Lines 818-825). In addition the window's "dirty" field in the document record is set to false (Line 827) and InvalRect is called to force a redraw of the window's content region (Line 829).

The function DoOpenCommand

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

Line 851 presents the Open dialog box, which will display the file types established at Lines 848-849. The sfType field of the StandardFileReply record "filled-in" by StandardGetFile contains the file type of the file selected by the user (853) and the sfFile field contains the file system specification. If the user clicks the OK button (Line 855), these are passed to the application-defined function DoOpenFile (Line 856).

The procedure 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 at Line 872 is immaterial. The document type 'TEXT' is passed in this instance simply to keep DoNewDocWindow happy.

The functions HasGotRequiredParams, DoOpenDocsEvent, and DoOpenAppEvent

The handlers for the required Apple events are essentially identical to those in the demonstration program at Chapter 8 - Required Apple Events.

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, Line 927 calls FSpGetFInfo to return the Finder information from the volume catalog entry for the file relating to the specified FSSpec record. The fdType field of the FInfo record "filled-in" by FSpGetFInfo contains the file type. This, together with the FSSpec record, is then passed in the call to DoOpenFile at Line 930. (DoOpenFile is also called when the user chooses Open from the File menu.)

Most programs should simply open a new untitled window on receipt of an Open Application event. Accordingly, DoOpenAppEvent (Lines 955-968) simply calls the same function (DoNewCommand) as is called when the user chooses New from the File menu.

The procedure DoMakeWindowDirty

DoMakeWindowDirty is called when the user chooses the Make Window "Dirty" item in the Demonstration menu. Changing the content of the in-memory version of a file is only simulated in this program. A diagonal line is drawn in the window (Lines 983-984) and the windowDirty field of the document record is set to true (Line 986).

The procedure DoAdjustMenus

DoAdjustMenus is invoked when the user clicks on the menu bar or presses a Command-key equivalent. Basically, the File menu items Close, Save, Save As, and Revert to Saved are disabled whenever no windows are open. In addition:

- The Revert to Saved item is disabled whenever there is no document associated with the front window. (As shown, one way to test whether the window has a document is to check both the pictureHdl field of the document record and the teLength field of the text edit record (Line 1015).)

- The Demonstration menu item is disabled when the windowDirty field of the document record associated with the front window indicates that the window is "dirty" (Lines 1019-1022).

The procedure DoFileMenu

DoFileMenu handles File menu choices. In each case, the relevant application-defined routine is called and, if that function returns an error, the application-defined function DoError is called. Note that, in the case of the Quit command (Line 1091), gDone is set to true after DoQuitCommand returns (assuming the user didn't cancel), thus causing the program to terminate.

The procedures DoMenuChoice, DoUpdate, DoMouseDown, DoEvents, and EventLoop

DoEvents, EventLoop, DoMouseDown and DoUpdate perform such low-level and Operating System event processing as is necessary for the satisfactory execution of the demonstration aspects of the program.

DoMenuChoice performs the initial handling of menu choices.

The function DoQuitAppEvent

Quit Application events are handled at Lines 1306-1326. The while loop entered at Line 1315 repeats for each open window. Line 1317 calls doCloseCommand which, 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 (Lines 1318-1319). 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 (Lines 1320-1321). When the while loop eventually exits, gDone is set to true (Line 1322), causing the program to terminate.

The procedure 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.)

The main program block

The main function initialises the system software managers (Line 1363), assigns the file reference number of the application's resource fork (which is opened automatically at application launch) to a global variable (Line 1368), sets up the menus (Lines 1372-1382), installs the required Apple event handlers (Line 1386), and enters the main event loop (Line 1390).