9

QuickTime Movies and File Handling

AT THIS POINT, YOU KNOW ALL about interface elements such as windows, controls, and menus, and you have a firm grasp on how your program recognizes and handles a variety of types of events. So it's on to the fun stuff. In this chapter, you'll see how your program opens and plays a QuickTime movie. QuickTime is movie-playing software that is part of the system software of every Macintosh in your target audience.

It's possible for a program to cause a QuickTime movie to spring forth from (seemingly) nowhere. However, it's more likely that a movie-playing application will enable the user to select the file that holds the movie to play. Giving the user the power to open a QuickTime movie file, or any other type of file, involves the Open dialog box. We'll look at the Open dialog box first in this chapter.

Files and Navigation Services

A *file* is a sequence of bytes stored on physical media, such as a hard drive, and a *directory* is another name for a folder. A *volume* can be an entire physical storage device or it can be part of the device (the result of formatting the device to consist of multiple volumes). For a program to access a file, it needs to know the file's name, the directory in which the file is located, and the volume on which that directory resides. In certain circumstances, a program that's to open a file includes these values (the file name and location) directly within its code, but that's a scenario few programs use. In addition,

this hard-coding of file information prevents the user from choosing what file to open, and it also sets up an application failure should the user move or delete the sought-after file.

A better way to handle the situation is to call Navigation Services routines to make use of the Open dialog box. By displaying the Open dialog box, you enable a user to select the file to open. Handling file opening in this way also forces the system to do the work of determining a file's name and location, and it leaves it to the system to convey this important file information to your program.

The Open dialog box provides the user with a standard interface for opening a file. This same Open dialog box is used by any real-world application. You can see it by choosing Open from the File menu of programs such as Apple's TextEdit or by looking at Figure 9.1.

Navigation Services is part of the Carbon API that makes is possible for your programs to include standard dialogs such as the Open dialog box. In addition, it is an important and useful part of the Carbon API. It routines provide interface consistency for the user and removes the burden of file location determination from the programmer. In this chapter, you'll see how to make use of Navigation Services, so brace yourself for a barrage of information about key Navigation Services routines.

Implementing an Open Dialog Box

You'll make use of a number of Navigation Services routines to display and handle an Open dialog box that is similar to the one TextEdit and other Mac OS X applications use. To do that, your code will perform the following:

1. Create and display the standard Open dialog box.
2. Become aware of the user's action in the Open dialog box.
3. Respond to the user's action (for instance, open the appropriate file if the user clicks the Open button).
4. Clean up by disposing of the Open dialog box when appropriate.

The overall look and behavior of an Open dialog box usually is the same. Such a dialog box includes Cancel and Open buttons and a list view of the folders and files on the user's machine. The general behavior of this type of dialog box is the same from one implementation to another as well; the user navigates through the file list, clicks the name of a file to open within the list, and then clicks the Open button to open the selected file. To promote this consistent look and behavior, Navigation Services defines the `NavDialogCreationOptions` data structure as the following:

```
struct NavDialogCreationOptions {
   UInt16                version;
   NavDialogOptionFlags  optionFlags;
   Point                 location;
   CFStringRef           clientName;
   CFStringRef           windowTitle;
```

```
    CFStringRef           actionButtonLabel;
    CFStringRef           cancelButtonLabel;
    CFStringRef           saveFileName;
    CFStringRef           message;
    UInt32                preferenceKey;
    CFArrayRef            popupExtension;
    WindowModality        modality;
    WindowRef             parentWindow;
    char                  reserved[16];
};
typedef struct NavDialogCreationOptions NavDialogCreationOptions;
```

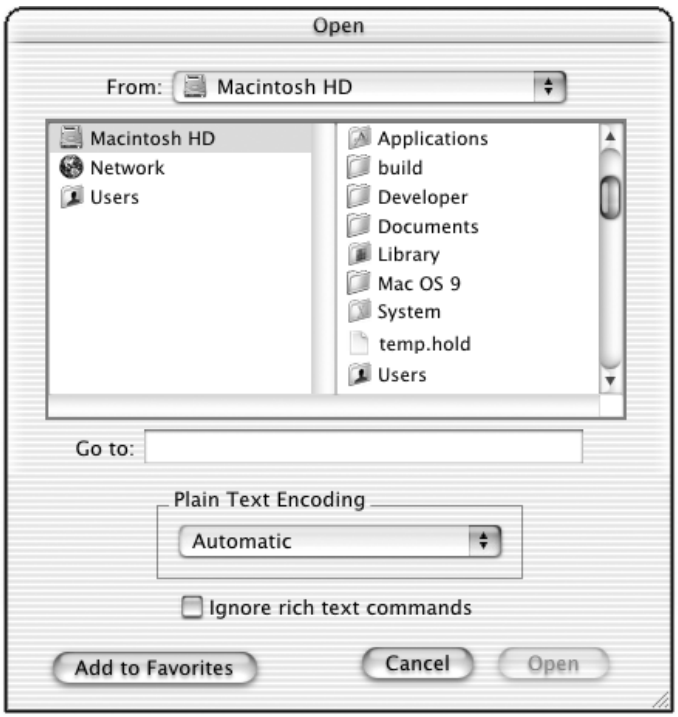

Figure 9.1 A typical Open dialog box (as displayed in TextEdit).

The `NavDialogCreationOptions` structure defines the features (such as size and location) of an Open dialog box. The Navigation Services routine `NavGetDefaultDialogCreationOptions` is used to fill the fields of a `NavDialogCreationOptions` structure with default values. Use this routine by declaring a variable of type `NavDialogCreationOptions` and then passing that variable's address as the routine's one argument:

```
OSStatus                  err;
NavDialogCreationOptions  dialogAttributes;

err = NavGetDefaultDialogCreationOptions( &dialogAttributes )
```

After setting the values of the members of a structure to default values, you can customize the structure by changing the value of any individual member. For instance, to make the Open dialog box take over the application and disallow other application actions to take its place, the value of the dialog box's `NavDialogCreationOptions modality` member can be set to the Apple-defined constant `kWindowModalityAppModal`:

```
dialogAttributes.modality = kWindowModalityAppModal;
```

You've seen how a program includes an application-defined event handler routine that's associated with a window or other object. The Open dialog box also needs an application-defined event handler routine associated with it. This event handler will be called by the system when the user dismisses the Open dialog box. Navigation Services creates, displays, and runs the Open dialog box, but it is this event handler that should perform the actual work of opening a user-selected file. Like other event handlers, this Open dialog box event handler can have a name of your choosing, but it must include arguments of specific types. Here's the prototype for such a routine:

```
pascal void MyNavEventCallback(
                              NavEventCallbackMessage callBackSelector,
                              NavCBRecPtr             callBackParms,
                              void*                   callBackUD );
```

In a moment, you'll pass a pointer to this event handler to the Navigation Services routine that creates the Open dialog box. The pointer should be of type `NavEventUPP`. The *UPP* in NavEventUPP stands for *universal procedure pointer*, which is a pointer that is capable of referencing procedures, or routines, in different executable formats. In this case, a `NavEventUPP` can point to a routine that is in native Mac OS X executable format or in pre-Mac OS X executable format. You'll also need this pointer elsewhere in your program, so declaring this pointer globally makes sense:

```
NavEventUPP   gNavEventHandlerPtr;
```

Use the Navigation Services routine `NewNavEventUPP` to set this routine pointer variable to point to the Open dialog box event handler:

```
gNavEventHandlerPtr = NewNavEventUPP( MyNavEventCallback );
```

Now it's time to make a call to the Navigation Services routine `NavCreateGetFileDialog` to create the Open dialog box. This routine requires seven arguments, many of which can typically get set to `NULL`. Here's the function prototype:

```
NavCreateGetFileDialog(
   const NavDialogCreationOptions *   inOptions,
   NavTypeListHandle                  inTypeList,
   NavEventUPP                        inEventProc,
   NavPreviewUPP                      inPreviewProc,
   NavObjectFilterUPP                 inFilterProc,
   void *                             inClientData,
   NavDialogRef *                     outDialog );
```

Using the previously declared `dialogAttributes` and `gNavEventHandlerPtr` variables, here's how a call to `NavCreateGetFileDialog` could look:

```
NavDialogRef   openDialog;

err =  NavCreateGetFileDialog( &dialogAttributes, NULL,
                               gNavEventHandlerPtr, NULL, NULL,
                               NULL, &openDialog );
```

The `inOptions` parameter is a pointer to the set of Open dialog box features that was returned by a prior call to `NavGetDefaultDialogCreationOptions`. In the preceding code snippet, `dialogAttributes` holds that set of default values, with the exception of the modality that was altered after `NavGetDefaultDialogCreationOptions` was called.

The `inTypeList` is a list of file types to display in the Open dialog box's browser; pass `NULL` to display all file types.

The `inEventProc` parameter is the procedure pointer that points to the Open dialog box's event handler routine. In the preceding snippet, the global UPP variable `gNavEventHandlerPtr`, which was assigned its value from a call to `NewNavEventUPP`, is used.

The next three arguments each can be set to `NULL`. The `inPreviewProc` parameter is a pointer to a custom file preview routine. The `inFilterProc` parameter is a pointer to a custom file filter routine. The `inClientData` parameter is a value that gets passed to either of the just-mentioned custom routines (if present). The preceding snippet uses `NULL` for each of these three arguments.

The last argument is a pointer to a variable of type `NavDialogRef`. After `NavCreateGetFileDialog` executes, this argument will hold a reference to the newly created Open dialog box.

`NavCreateGetFileDialog` creates an Open dialog box, but it doesn't display or control it. To do those chores, call the Navigation Services routine `NavDialogRun`:

```
err = NavDialogRun( openDialog );
```

`NavDialogRun` handles the user's interaction with the Open dialog box, so you don't need to write any code to follow the user's actions as he or she uses the dialog box to browse for a file to open. When the user clicks the Cancel or Open button, the application-defined event handler associated with this Open dialog box is called. In doing this, Navigation Services passes on information about the event that initiated the event handler call.

As you'll see a little later in this chapter, the event handler takes care of the opening of the selected file and the dismissing of the Open dialog box. Control then returns to the code that follows the call to `NavDialogRun`. That code should look something like this:

```
if ( err != noErr )
{
   NavDialogDispose( openDialog );
   DisposeNavEventUPP( gNavEventHandlerPtr );
}
```

If `NavDialogRun` completes without an error, your work is done. If there *was* an error, the variable `err` will have a nonzero (non-`noErr`) value. Your code should call the Navigation Services routines `NavDialogDispose` to dispose of the Open dialog box reference and `DisposeNavEventUPP` to dispose of the pointer to the Open dialog box event handler.

Whew. That covers the process of displaying and running the Open dialog box. Now it's time to take a look at all the code as it might appear in an application-defined routine that is used to enable a user to choose a file to open:

```
void DisplayOpenFileDialog( void )
{
   OSStatus                  err;
   NavDialogRef              openDialog;
   NavDialogCreationOptions  dialogAttributes;

   err = NavGetDefaultDialogCreationOptions( &dialogAttributes );

   dialogAttributes.modality = kWindowModalityAppModal;

   gNavEventHandlerPtr = NewNavEventUPP( MyNavEventCallback );

   err =  NavCreateGetFileDialog( &dialogAttributes, NULL,
                                  gNavEventHandlerPtr, NULL, NULL,
                                  NULL, &openDialog );

   err = NavDialogRun( openDialog );

   if ( err != noErr )
   {
      NavDialogDispose( openDialog );
      DisposeNavEventUPP( gNavEventHandlerPtr );
   }
}
```

Open Dialog Box Event Handler

After the user of an Open dialog box makes a final decision (by clicking the Cancel or Open button), the Open dialog box event handler is automatically invoked. When the system invokes this handler, the system passes information about the event initiated by the user's action:

```
pascal void MyNavEventCallback(
                               NavEventCallbackMessage callBackSelector,
                               NavCBRecPtr             callBackParms,
                               void*                   callBackUD )
```

Your event handler uses the information in the `callBackSelector` argument to determine the action with which to deal. The bulk of the event handler consists of a `switch` statement that determines which of the primary dialog box-related tasks needs handling:

```
switch ( callBackSelector )
{
```

```
    case kNavCBUserAction:
       // further determine which action took place (open or save)
       // handle the action (open or save selected file)
       break;

    case kNavCBTerminate:
       // clean up after the now-dismissed dialog
       break;
}
```

The main two tasks handled in the `switch` consist of a user action (`kNavCBUserAction`), such as the request to open a file, and the memory clean up (`kNavCBTerminate`), which is in response to the dismissal of the dialog box.

To respond to a user action, call the Navigation Services routine `NavDialogGetReply`. Pass this routine a reference to the dialog box that initiated the event and a pointer to a reply record. `NavDialogGetReply` will fill the reply record with information about the user's action (such as the file to open). The `context` field of the event handler argument `callBackParms` holds the dialog reference. Declare a variable of type `NavReplyRecord` to be used as the reply record:

```
OSStatus         err;
NavReplyRecord   reply;
NavUserAction    userAction = 0;

err = NavDialogGetReply( callBackParms->context, &reply );
```

Now call `NavDialogGetUserAction`, passing this routine a reference to the affected dialog box. Once again, the context field of the `callBackParams` event handler argument is used. `NavDialogGetUserAction` tells your program the exact action the user took. In the case of an Open dialog box, you're looking for a user action of `kNavUserActionOpen`. Note that similar code is used to handle a Save dialog, and in such a case, you'd look for a user action of `kNavUserActionSaveAs`. Finish with a call to `NavDisposeReply` to dispose of the reply record.

```
userAction = NavDialogGetUserAction( callBackParms->context );

switch ( userAction )
{
   case kNavUserActionOpen:
   // open file here using reply record information
   break;
}
err = NavDisposeReply( &reply );
```

Note

The preceding code snippet includes one very vague comment. Obviously, some code needs to actually open the user-selected file, yet I've waved that chore off with a single comment. That's because the particulars of opening a file are specific to the type of file to open; a move file, a graphics file, and an application-defined file all require different code to be transformed from data on media to data in memory and finally to information displayed in a window. Later in this chapter, we'll jump into the general steps, and the detailed code, for opening one type of file: a QuickTime movie file.

You can put the just-described Open dialog box event handler code into a routine that looks like the one shown here:

```
pascal void MyOpenDialogEventCallback(
                              NavEventCallbackMessage callBackSelector,
                              NavCBRecPtr             callBackParms,
                              void*                   callBackUD )
{
   OSStatus         err;
   NavReplyRecord   reply;
   NavUserAction    userAction = 0;

   switch ( callBackSelector )
   {
      case kNavCBUserAction:
         err = NavDialogGetReply( callBackParms->context, &reply );
         userAction = NavDialogGetUserAction( callBackParms->context );

         switch ( userAction )
         {
            case kNavUserActionOpen:
            // open file here using reply record information
            break;
         }
         err = NavDisposeReply( &reply );
         break;

      case kNavCBTerminate:
         NavDialogDispose( callBackParms->context );
         DisposeNavEventUPP( gNavEventHandlerPtr );
         break;
   }
}
```

The `MyOpenDialogEventCallback` routine is generic enough that it should work, with very little alteration, in your own file-opening program. Now all you need to do is replace the routine's one comment with a call to an application-defined function designed to open a file of the appropriate type. In the next section, you see how to write such a routine. The code for the application-defined function `OpenOneQTMovieFile` opens a QuickTime movie file. The OpenPlayMovie example then uses the `MyOpenDialogEventCallback` routine with a call to `OpenOneQTMovieFile`.

QuickTime Movies

A sound knowledge of the fundamentals of developing an interface for your Mac OS X program is of great importance, but you didn't choose to learn about Mac programming for the sole purpose of creating windows that include a few buttons. You most certainly also want to know how your own program can include at least *some* multimedia capabilities.

Unfortunately, Mac OS X programming for sound playing, sound recording, and smooth animation are worthy of their own programming book. So, what can I show you in just half a chapter? Well, I can show you one multimedia topic that best showcases Mac OS X multimedia in action: QuickTime. By giving your program the ability to play QuickTime movies, you can add high-resolution graphics, animation, and sound playing to your program.

> **Note**
> QuickTime is now cross-platform software, but it started out as an extension of Mac-only system software.

In this section, you see how to use the previously discussed Navigation Services routines to present the user with an Open dialog box that lets him or her choose a QuickTime movie file to open. After that file is open, you use Carbon API routines (which are grouped into the Movie Toolbox area of the Carbon API) to play the movie.

Opening a QuickTime Movie File

This chapter's "Files and Navigation Services" section provides all the details for presenting the user with a standard Open dialog box. It also shows how to respond to a user's selection of a file that is listed in that dialog box.

In this part of the chapter, you'll be using that information to give the user the power to pick a QuickTime movie file to display. Specifically, I'll jump into descriptions of the techniques and Movie Toolbox routines that your program will use to get QuickTime movie file data that exists on the user's disk into a format that's ready to play as a movie in a window. The result will be an application-defined routine named `OpenOneQTMovieFile`. Then, after you've developed this routine, you can insert it into the `kNavUserActionOpen` case label section of the `switch` statement in the `MyOpenDialogEventCallback` routine that was developed earlier in this chapter.

Transferring Movie File Data to Memory

Scan back just a bit in this chapter and you'll see the heading "Opening a QuickTime Movie File." Look ahead a little and you'll see the heading "Playing a QuickTime Movie." In broad terms, these are the two steps a program performs so that a user can view a movie. However, each step is more involved that it would first appear. For instance, in the case of opening a movie file, what's actually taking place is the opening of that file (so its data can be accessed), the copying of that file's movie data content into memory (where it can be referenced by the application), and the closing of the file (because its contents are no longer needed). The goal of what's loosely described as the opening of a file is actually the transferring (or copying) of a file's data into memory.

To open a file, your program needs to know the file's name and location. If the user selected the file in the Open dialog box, that dialog box's event handler gets the required information from the `NavReplyRecord` variable. Recall from this chapter's

"Open Dialog Box Event Handler" section that the Open dialog box event handler called `NavDialogGetReply` to fill a `NavReplyRecord` with information about the user-selected file to open:

```
NavReplyRecord   reply;

err = NavDialogGetReply( callBackParms->context, &reply );
```

With years of computer programming experience comes an appreciation for a programming task as simple as adding two numbers; the job's simplicity ensures there's little or no chance of error. This is in contrast to a task such as file handling, which can be fraught with peril! The task involves selecting a file, opening it, copying its contents to memory, and then accessing that memory to make use of the data within. One flipped bit in this process can really play havoc on a program or even the drive itself!

In an attempt to avoid intimacy with the debugger, file-handling code often makes judicious use of error checking. To increase the explanation-to-code ratio in this book, I've provided descriptions of some basic error-handling techniques in Chapter 2, "Overview of Mac OS X Programming," and then for the most part, kept error-handling code to a minimum in the subsequent chapters. Now, however, is no time to be stingy with error checking, so in upcoming snippets, you'll see a little extra precautionary code, starting right here:

```
OSStatus    err;
AEDesc      newDescriptor;
FSRef       movieRef;

err = AECoerceDesc( &reply->selection, typeFSRef, &newDescriptor );

err = AEGetDescData( &newDescriptor, ( void * )( &movieRef ),
                     sizeof( FSRef ) );
```

The Apple Event Manager routine `AECoerceDesc` accepts data of one type (the first argument), manipulates it to another type (specified by the second argument), and saves the results in a new variable (the third argument). The usage of this routine verifies that the `reply` variable that holds the user-selected file is in the format of an `FSRef`. After the call to `AECoerceDesc` completes, your program is assured of having an `FSRef` within the variable `newDescriptor`. The Apple Event Manager routine `AEGetDescData` then is called to retrieve the `FSRef` from the `newDescriptor` variable.

At this point, the program has an `FSRef` (the variable `movieRef`) that holds information about the user-selected file. Thus, we're *almost* ready to open the QuickTime movie file. However, we need to make one quick detour. Some of the Carbon API routines are older (they existed as original Macintosh Toolbox API routines), and some are newer (they were created to handle Mac OS X tasks for which no original Macintosh Toolbox routine existed). The newer file-handling Carbon API routines that require information about a file accept that information in the form of an argument of type `FSRef`. In contrast, original file-handling Toolbox routines that became part of the Carbon API look for this same information in the form of an argument of type

FSSpec. In addition, opening a QuickTime movie file requires the use of one of these older FSSpec-accepting routines. Fortunately, for situations such as this, the routine FSGetCatalogInfo function can be used to convert an FSRef to an FSSpec:

```
FSSpec   userFileFSSpec;

FSGetCatalogInfo( &movieRef, kFSCatInfoNone, NULL, NULL,
                  &userFileFSSpec, NULL );
```

FSGetCatalogInfo is a workhorse of a utility routine in that it can be used to obtain all sorts of information about a catalog file. (A catalog file is a special file used to keep information about all the files and directories on a volume.) You can use FSGetCatalogInfo to obtain information such as the reference number of the volume on which a file resides or the parent directory ID of a file. You also can use FSGetCatalogInfo to simply obtain an FSSpec for a file for which you already have an FSRef. That's what I'm interested in here. Of most importance in this usage of FSGetCatalogInfo is the first argument, which is a pointer to the FSRef to convert, and the fifth argument, which is a pointer to an FSSpec variable that FSGetCatalogInfo is to fill with the file system specification. The only other non-NULL value is the second argument. This argument normally is used to specify which of many pieces of information about a file or directory are to be returned. I don't need any of this information, so the constant kFSCatInfoNone is used here.

Now it's time to open the file. The Movie Toolbox routine OpenMovieFile does that. The first OpenMovieFile argument is a file system specification. You can use the one returned by the call to FSGetCatalogInfo. After OpenMovieFile opens the specified filem it provides your program with a reference number for that file. That reference number is your program's means of (you guessed it) referring to that file in subsequent calls to Movie Toolbox routines. The next argument is a pointer to a variable in which OpenMovieFile places this reference value. The last argument is a permission level for the opened file. A program that opens a movie for playing but that won't enable the altering of the movie contents should use the constant fsRdPerm.

```
OSErr     err;
FSSpec    userFileFSSpec
short     movieRefNum;

err = OpenMovieFile( &userFileFSSpec, &movieRefNum, fsRdPerm );
```

Caution

Besides fsRdPerm, other permission constants include fsWrPerm (to enable writing) and fsRdWrPerm (to enable reading and writing). In my simple examples, the permission level isn't crucial. That is, you can change it to, say, fsRdWrPerm, and the user still won't be able to cut any frames from an opened movie. However, in your full-blown application, permissions might be of importance. If your program includes a functioning Edit menu that supports the cutting and pasting of multiple data types, you might not want to give the user the ability to alter the frames of a movie. In such an instance, you'll want to make sure that movie files are opened with the fsRdPerm constant rather than with one of the constants that enables file writing.

After opening a movie file, that file's data needs to be loaded into memory. A call to the Movie Toolbox routine `NewMovieFromFile` does this:

```
Movie     movie = NULL;
short     movieResID = 0;

err = NewMovieFromFile( &movie, movieRefNum, &movieResID,
                        NULL, newMovieActive, NULL );
```

After `NewMovieFromFile` completes, the first argument holds a reference to the movie (a variable of type `Movie`). To create this movie, `NewMovieFromFile` needs the movie file reference number that was returned by the prior call to `OpenMovieFile`. You should pass this as the second argument. `NewMovieFromFile` also needs the ID of the movie data in the file in question. Although a single file typically holds one movie, it can hold multiple movies. Thus, it's necessary to specify which of a file's movies is to be used. A value of 0 as the third argument tells `NewMovieFromFile` to use the first movie in the file. Thus, even if there is only one movie in the file, this value of 0 does the job.

When `NewMovieFromFile` exits, it fills in the fourth argument (`movieName`) with the name of the movie resource that was used to create the movie. Note that this isn't the name of the file that holds the movie; it's the name of a resource within the file. That's usually not of importance, so your program can pass `NULL` here. The fifth argument is used to provide supplemental information to `NewMovieFromFile`. Using the constant `newMovieActive` specifies that the new movie should be active; a movie needs to be active for it to be played. The last argument tells whether `NewMovieFromFile` had to make any changes to the data in the file. This shouldn't occur, so again a value of `NULL` typically suffices.

The call to `OpenMovieFile` opened the file in preparation for access to it. `NewMovieFromFile` is the routine that accessed the file. Now, with the movie data safe in memory and a `Movie` variable referencing that data, the file can be closed:

```
CloseMovieFile( movieRefNum );
```

`CloseMovieFile` needs to know which file to close. The reference number returned by `OpenMovieFile` provides that information.

Displaying a Movie in a Window

At this point, a movie is in memory and accessible by way of a `Movie` variable. Now the movie needs to be associated with a window. There's nothing special about a window that holds a movie; you just create a new window resource in your program's main.nib file. You can make the window any size you want. Your code resizes this window to match the size of the movie that eventually gets displayed within the window. With the window resource defined, include the standard window-creation code in your code:

```
WindowRef   window;
OSStatus    err;
IBNibRef    nibRef;
```

```
err = CreateNibReference( CFSTR("main"), &nibRef );
err = CreateWindowFromNib( nibRef, CFSTR("MovieWindow"), &window );
DisposeNibReference( nibRef );
```

Now, for the link between the movie and the window, call `SetPortWindowPort` to make the window's port the active port. Then, call the Movie Toolbox routine `SetMovieGWorld` to associate the movie with the currently active port:

```
SetPortWindowPort( window );

SetMovieGWorld( movie, NULL, NULL );
```

The `GWorld` in `SetMovieGWorld` refers to a graphics world, which is a complex memory drawing environment used in the preparation of images before their onscreen display. The first `SetMovieGWorld` argument is the movie to associate with a port. The second argument is the port; pass `NULL` here to tell `SetMovieGWorld` to associate the movie with the current port, which is the window named in the call to `SetPortWindowPort`. The last argument is a handle to a `Gdevice`, which is a structure describing a graphics device. A value of `NULL` here tells `SetMovieGWorld` to use the current device.

Now determine the size of the open movie and use those coordinates to resize the window to match the movie size:

```
Rect   movieBox;

GetMovieBox( movie, &movieBox );
OffsetRect( &movieBox, -movieBox.left, -movieBox.top );
SetMovieBox( movie, &movieBox );
SizeWindow( window, movieBox.right, movieBox.bottom, TRUE );
ShowWindow( window );
```

Pass `GetMovieBox` a movie and the routine returns a rectangle that holds the size of the movie. This might be all you need, or it might not be. Although the returned rectangle does hold the size of the movie, it's possible that the top and left coordinates of this rectangle each might not be 0. In such a case, looking at `movieBox.right` for the movie's width and `movieBox.bottom` for the movie's height would provide erroneous information. For instance, a `movieBox.left` value of 50 and a `movieBox.right` value of 200 means that the movie has a width of 150 pixels. A call to the QuickDraw routine `OffsetRect` simply offsets the `movieBox` rectangle such that its left and top coordinates each have a value of 0. A call to `SetMovieBox` makes the new, offset values the boundaries for the rectangle that defines the size of the movie.

Although the movie rectangle has been adjusted, the window that's to display the movie has not. A call to `SizeWindow` does that. Pass `SizeWindow` the window to resize, along with the new width and height to use in the size change. The last argument is a Boolean value that tells whether an update event should be generated. The call to `ShowWindow` finally reveals the movie-holding window to the user.

To ensure that the window displays a frame of the movie, call `MoviesTask`. This Movie Toolbox routine does just that. Pass the movie to use in the frame display as the

first argument and a value of 0 as the second argument. This 0 value tells `MoviesTask` to service (update) each active movie. If your program can display more than one movie at a time, `MoviesTask` will jump to each open movie, displaying one new frame in each. Precede the call to `MoviesTask` with a call to `GoToBeginningOfMovie`. This Toolbox routine rewinds the movie to its first frame. Although a newly opened movie will most likely be set to the movie's first frame, a call to this routine ensures that that will be so:

```
GoToBeginningOfMovie( movie );
MoviesTask( movie, 0 );
```

Playing a QuickTime Movie

The movie's now open and displayed in a window. Let's play it from start to finish:

```
StartMovie( movie );

do
{
   MoviesTask( movie, 0 );
} while ( IsMovieDone( movie ) == FALSE );
```

Contrary to its name, `StartMovie` doesn't start a movie. Instead, it prepares the specified movie for playing by making the movie active and setting the movie's playback rate. To actually play a movie, call `MoviesTask` within a loop. Each call to `MoviesTask` plays a frame of the movie. Because your program won't know how many frames are in the movie to play, rely on a call to the Movie Toolbox routine `IsMovieDone` to determine when the frame-playing loop should terminate. Pass `IsMovieDone` a movie and the routine returns a value of `TRUE` if the last frame has been reached or `FALSE` if there's one or more frames left to play.

> **Note**
>
> Related to the running of a movie is the movie controller. It is the thin, three-dimensional control that runs along the bottom of a window displaying a QuickTime movie. The movie controller is under the user's control, and it enables the user to run, pause, or step forward or backwards through the movie displayed in the window. For more information on movie controllers, see the URL listed at the end of this chapter.

OpenPlayMovie Program

The purpose of OpenPlayMovie is to demonstrate how the Navigation Services routines are used to display the standard Open dialog box. It also shows how to respond to a user-selected file when that file is a QuickTime movie.

OpenPlayMovie starts by opening a window that includes a single line of text, as shown in Figure 9.2. When you follow that window's instructions, you see the

standard Open dialog box. Use the file lists to move about your drive or drives to find a QuickTime movie. When you click the Open dialog box's Open button, a new window displaying the first frame of the movie appears. Choose Play Movie from the Movie menu and the movie plays from start to finish. You can choose Play Movie as often as you wish.

Figure 9.2 Windows displayed by the OpenPlayMovie program.

Nib Resources

The main.nib window in the project's main.nib file includes two windows, as shown in Figure 9.3. By default, Interface Builder sets a new window to be resizable, and it gives the window a small resize control in the lower-right corner of the window. The OpenPlayMovie program eliminates this resize control from the movie-playing window. The control would obscure a small part of one corner of the movie if it were present.

You can use Interface Builder to set a window so that it can't be resizable. To do that, click the MovieWindow window in main.nib (see Figure 9.3), choose Show Info from the Tools menu, display the Attributes pane, and then *uncheck* the Resizable checkbox. While you're there, uncheck both the Close and the Zoom checkboxes so that the window won't be closeable or zoomable.

The Movie menu includes two items: Open Movie and Play Movie. The Open Movie item has a command of `Opmv`. Assign it that command from the Attributes pane of the item's Info window. The Play Movie item has a command of `PLmv`. Play Movie

item is initially disabled. Click that item, choose Show Info from the Tools menu, and, from the Attributes pane, uncheck the Enabled checkbox. Because the program's code will be accessing the Movie menu, this menu needs an ID. You can give the Movie menu a menu ID of 5 by clicking Movie in the menu window, choosing Show Info from the Tools menu, and entering 5 in the Menu ID field.

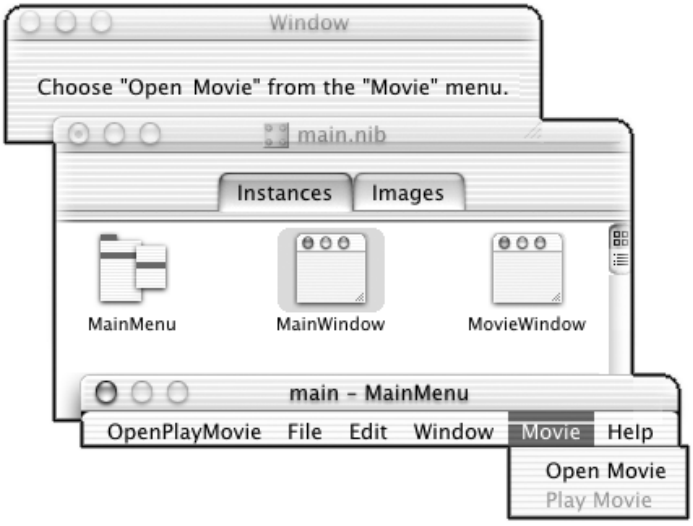

Figure 9.3 The OpenPlayMovie nib resources.

Source Code

The QuickTime function prototypes aren't included in a project by default, so you'll need to include QuickTime.h along with Carbon.h:

```
#include <Carbon/Carbon.h>
#include <QuickTime/QuickTime.h>
```

Define a constant to match the commands assigned to the Open Movie and Play Movie menu items in the nib resource. Also, define constants to match the Movie menu ID and the menu placement of the two items in the Movie menu:

```
#define  kOpenMovieCommand     'OPmv'
#define  kPlayMovieCommand     'PLmv'
#define  kMovieMenuID             5
#define  kMovieMenuOpenItemNum    1
#define  kMovieMenuPlayItemNum    2
```

OpenPlayMovie declares three global variables. The procedure pointer `gNavEventHandlerPtr` is used in setting up the Open dialog box event handler, `gMovie` will reference the movie after it's opened, and `gMovieMenu` will hold a handle to the

Movie menu so that the menu's items can be enabled and disabled:

```
NavEventUPP   gNavEventHandlerPtr;
Movie         gMovie = NULL;
MenuHandle    gMovieMenu;
```

Almost all the `main` routine is the same as in past examples. Additions of note include a call to `EnterMovies` (a Movie Toolbox initialization routine that's required before a program makes use of other Movie Toolbox routines) and a call to `GetMenuHandle` (to obtain a handle to the Movie menu):

```
int main( int argc, char* argv[] )
{
   IBNibRef          nibRef;
   WindowRef         window;
   OSStatus          err;
   EventTargetRef    target;
   EventHandlerUPP   handlerUPP;
   EventTypeSpec     appEvent = { kEventClassCommand,
                                  kEventProcessCommand };

   EnterMovies();

   // set up menu bar, open window, install event handler

   gMovieMenu = GetMenuHandle( kMovieMenuID );

   RunApplicationEventLoop();

   return( 0 );
}
```

OpenPlayMovie demonstrates some simple menu adjustment techniques that make it possible to force the program to enable only one movie to be opened. When the program launches, the Open Movie item is enabled and the Play Movie item is disabled, as specified in the menu resource in the nib file. If a menu item doesn't make sense at a particular moment in the running of a program, it should be disabled. When the program launches, no movie is open, so the Play Movie item isn't applicable. That's why it's initially disabled. The program enables a user to select a movie to open, so the Open Movie item starts out enabled. The toggling of the state of these two items takes place in the application's event handler.

The following snippet comes from `MyAppEventHandler` and shows the code that responds to a command issued by the user's choosing of the Open Movie menu item:

```
case kOpenMovieCommand:
   DisplayOpenFileDialog();
   if ( gMovie != NULL )
   {
      DisableMenuItem( gMovieMenu, kMovieMenuOpenItemNum );
      EnableMenuItem( gMovieMenu, kMovieMenuPlayItemNum );
   }
   result = noErr;
   break;
```

Handling an Open Movie menu item selection begins with a call to the application-defined `DisplayOpenFileDialog` routine. The global `Movie` variable `gMovie` was initialized to a value of `NULL` to signify that no movie is open. If the user opens a movie, `gMovie` references that movie and will have a value other than `NULL`. In that case, `MyAppEventHandler` disables the Open Movie item and enables the Play Movie item. That makes it impossible for the user to attempt to open a second movie, and it makes possible the playing of the now-open movie. If the user clicks the Cancel button in the Open dialog box, `gMovie` will retain its `NULL` value and the two menu items will retain their initial state. This enables the user to again choose Open Movie to open a movie.

In your more sophisticated movie-playing programs, you might allow the display and playing of multiple movies. In that case, you can expand on the technique discussed here by allowing the closing of movie windows and the toggling of the Play Movie item from enabled to disabled when all such movie windows are closed. One way to do that is to intercept window-closing events. When a window closes, check whether it was the last movie window. (You could keep a global movie window counter that increments and decrements as movies are opened and closed.) In Chapter 3, "Events and the Carbon Event Manager," the MyCloseWindow example introduces the topic of window closing events. (The program sounds a beep when the user clicks a window's Close button.) In Chapter 4, "Windows," the MenuButtonCloseWindow example elaborates on this technique. Finally, in Chapter 6, "Menus," you learn how to enable and disable menu items.

If the user chooses Play Movie, the application-defined `PlayOneMovie` routine is called. Note that there's no need for any menu-item disabling or enabling here. If this item is enabled, it means a movie window is open and can be played. If no movie window is open, this item will be disabled and the `kPlayMovieCommand` can't be generated by the program!

```
case kPlayMovieCommand:
   PlayOneMovie( gMovie );
   result = noErr;
   break;
```

In response to the user's choosing Open Movie, the program calls `DisplayOpenFileDialog`. This application-defined routine was developed in this chapter's "Implementing an Open Dialog Box" section. The OpenPlayMovie source code listing (Example 9.1) shows this routine. The event handler, or callback routine, that `DisplayOpenFileDialog` installs is the application-defined routine `MyOpenDialogEventCallback`. (This is another routine discussed at length in the "Implementing an Open Dialog Box" section. Refer to those pages for more information on this callback function.) Here I'll point out that if the system invokes this routine with a user action of `kNavUserActionOpen`, the callback routine invokes the application-defined function `OpenOneQTMovieFile` to open the user-selected movie file.

The `OpenOneQTMovieFile` routine is basically a compilation of the code discussed in this chapter's "Transferring Movie File Data to Memory" section. `AECoerceDesc` makes

sure that the NavReplyRecord filled in by the Open dialog box is valid, AEGetDescData retrieves an FSRef from that reply record, and FSGetCatalogInfo converts the FSRef to an FSSpec for use in opening the movie file:

```
void  OpenOneQTMovieFile( NavReplyRecord *reply )
{
   AEDesc      newDescriptor;
   FSRef       movieRef;
   WindowRef   window;
   OSStatus    err;
   FSSpec      userFileFSSpec;
   IBNibRef    nibRef;

   err = AECoerceDesc( &reply->selection, typeFSRef, &newDescriptor );

   err = AEGetDescData( &newDescriptor, ( void * )( &movieRef ),
                        sizeof( FSRef ) );

   FSGetCatalogInfo( &movieRef, kFSCatInfoNone, NULL, NULL,
                     &userFileFSSpec, NULL );

   gMovie = GetMovieFromFile( userFileFSSpec );
```

The application-defined routine GetMovieFromFile (discussed next) opens the movie file and assigns gMovie a reference to the movie. A new window then is opened, its port is set to the current port, and the application-defined routine AdjustMovieWindow (discussed shortly) resizes the window and associates the movie with the window. OpenOneQTMovieFile ends with a call to AEDisposeDesc to dispose of the AEDisc created earlier in the routine:

```
err = CreateNibReference( CFSTR("main"), &nibRef );
   err = CreateWindowFromNib( nibRef, CFSTR("MovieWindow"), &window );
   DisposeNibReference( nibRef );

   SetPortWindowPort( window );

   AdjustMovieWindow( gMovie, window );

   AEDisposeDesc( &newDescriptor );
}
```

GetMovieFromFile is a short routine that makes three Movie Toolbox calls. OpenMovieFile opens the user-selected file. NewMovieFromFile loads the movie data to memory and returns a reference to the movie. CloseMovieFile closes the movie file. This chapter's "Transferring Movie File Data to Memory" section discusses each routine.

```
Movie  GetMovieFromFile( FSSpec userFileFSSpec )
{
   OSErr     err;
   Movie     movie = NULL;
   short     movieRefNum;
```

```
short     movieResID = 0;

   err = OpenMovieFile( &userFileFSSpec, &movieRefNum, fsRdPerm );

   err = NewMovieFromFile( &movie, movieRefNum, &movieResID,
                           NULL, newMovieActive, NULL );

   CloseMovieFile( movieRefNum );

   return ( movie );
}
```

`AdjustMovieWindow` combines the code discussed in this chapter's "Displaying a Movie in a Window" section to create a routine that calls `SetMovieGWorld` to associate the open movie with the recently opened window and to resize the window to match the size of the movie.

```
void  AdjustMovieWindow( Movie movie, WindowRef window )
{
   Rect   movieBox;

   SetMovieGWorld( movie, NULL, NULL );

   GetMovieBox( movie, &movieBox );
   OffsetRect( &movieBox, -movieBox.left, -movieBox.top );
   SetMovieBox( movie, &movieBox );

   SizeWindow( window, movieBox.right, movieBox.bottom, TRUE );
   ShowWindow( window );

   GoToBeginningOfMovie( gMovie );
   MoviesTask( gMovie, 0 );
}
```

At this point, a movie file has been opened and the first frame of the movie is displayed in a window. To play the movie, the user chooses Play Movie from the Movie menu. Doing that initiates a command that the application event handler handles by calling `PlayOneMovie`. This routine bundles the code discussed in this chapter's "Playing a QuickTime Movie" section, with the result being the playing of the movie from start to finish:

```
void  PlayOneMovie( Movie movie )
{
   GoToBeginningOfMovie( movie );

   StartMovie( movie );

   do
   {
      MoviesTask( movie, 0 );
   } while ( IsMovieDone( movie ) == FALSE );
}
```

Example 9.1 OpenPlayMovie Source Code

```
#include  <Carbon/Carbon.h>
#include  <QuickTime/QuickTime.h>

#define   kOpenMovieCommand     'OPmv'
#define   kPlayMovieCommand     'PLmv'
#define   kMovieMenuID              5
#define   kMovieMenuOpenItemNum     1
#define   kMovieMenuPlayItemNum     2

Movie           GetMovieFromFile( FSSpec userFileFSSpec );
void            AdjustMovieWindow( Movie movie, WindowRef window );
void            PlayOneMovie( Movie movie );
void            DisplayOpenFileDialog( void );
void            OpenOneQTMovieFile( NavReplyRecord *reply ) ;
pascal OSStatus MyAppEventHandler( EventHandlerCallRef handlerRef,
                                   EventRef event, void *userData );
pascal void     MyOpenDialogEventCallback(
                              NavEventCallbackMessage callBackSelector,
                              NavCBRecPtr             callBackParms,
                              void*                   callBackUD );

NavEventUPP   gNavEventHandlerPtr;
Movie         gMovie = NULL;
MenuHandle    gMovieMenu;


int main( int argc, char* argv[] )
{
   IBNibRef          nibRef;
   WindowRef         window;
   OSStatus          err;
   EventTargetRef    target;
   EventHandlerUPP   handlerUPP;
   EventTypeSpec     appEvent = { kEventClassCommand,
                                  kEventProcessCommand };

   EnterMovies();

   err = CreateNibReference( CFSTR("main"), &nibRef );
   err = SetMenuBarFromNib( nibRef, CFSTR("MainMenu") );
   err = CreateWindowFromNib( nibRef, CFSTR("MainWindow"), &window );
   DisposeNibReference( nibRef );

   ShowWindow( window );

   target = GetApplicationEventTarget();
   handlerUPP = NewEventHandlerUPP( MyAppEventHandler );
   InstallEventHandler( target, handlerUPP, 1, &appEvent, 0, NULL );

   gMovieMenu = GetMenuHandle( kMovieMenuID );
```

continues

Example 9.1 Continued

```
    RunApplicationEventLoop();

    return( 0 );
}


pascal OSStatus MyAppEventHandler( EventHandlerCallRef handlerRef,
                                   EventRef event, void *userData)
{
    OSStatus    result = eventNotHandledErr;
    HICommand   command;

    GetEventParameter( event, kEventParamDirectObject, typeHICommand,
                       NULL, sizeof (HICommand), NULL, &command);

    switch ( command.commandID )
    {
       case kOpenMovieCommand:
          DisplayOpenFileDialog();
          if ( gMovie != NULL )
          {
             DisableMenuItem( gMovieMenu, kMovieMenuOpenItemNum );
             EnableMenuItem( gMovieMenu, kMovieMenuPlayItemNum );
          }
          result = noErr;
          break;

       case kPlayMovieCommand:
          PlayOneMovie( gMovie );
          result = noErr;
          break;
    }
    return result;
}


void DisplayOpenFileDialog( void )
{
    OSStatus                    err;
    NavDialogRef                openDialog;
    NavDialogCreationOptions    dialogAttributes;

    err = NavGetDefaultDialogCreationOptions( &dialogAttributes );
    dialogAttributes.modality = kWindowModalityAppModal;

    gNavEventHandlerPtr = NewNavEventUPP( MyOpenDialogEventCallback );


    err = NavCreateGetFileDialog( &dialogAttributes, NULL,
```

```
                                gNavEventHandlerPtr, NULL, NULL,
                                NULL, &openDialog );

   err = NavDialogRun( openDialog );
   if ( err != noErr )
   {
      NavDialogDispose( openDialog );
      DisposeNavEventUPP( gNavEventHandlerPtr );
   }
}


pascal void MyOpenDialogEventCallback(
                              NavEventCallbackMessage callBackSelector,
                              NavCBRecPtr             callBackParms,
                              void*                   callBackUD )
{
   OSStatus         err;
   NavReplyRecord   reply;
   NavUserAction    userAction = 0;

   switch ( callBackSelector )
   {
      case kNavCBUserAction:
         err = NavDialogGetReply( callBackParms->context, &reply );
         userAction = NavDialogGetUserAction( callBackParms->context );
         switch ( userAction )
         {
            case kNavUserActionOpen:
               OpenOneQTMovieFile( &reply );
               break;
         }
         err = NavDisposeReply( &reply );
         break;

      case kNavCBTerminate:
         NavDialogDispose( callBackParms->context );
         DisposeNavEventUPP( gNavEventHandlerPtr );
         break;
   }
}


void  OpenOneQTMovieFile( NavReplyRecord *reply )
{
   AEDesc      newDescriptor;
   FSRef       movieRef;
   WindowRef   window;
   OSStatus    err;
   FSSpec      userFileFSSpec;
   IBNibRef    nibRef;
```

continues

Example 9.1 Continued

```
    err = AECoerceDesc( &reply->selection, typeFSRef, &newDescriptor );

    err = AEGetDescData( &newDescriptor, ( void * )( &movieRef ),
                         sizeof( FSRef ) );

    FSGetCatalogInfo( &movieRef, kFSCatInfoNone, NULL,
                      NULL, &userFileFSSpec, NULL );

    gMovie = GetMovieFromFile( userFileFSSpec );

    err = CreateNibReference( CFSTR("main"), &nibRef );
    err = CreateWindowFromNib( nibRef, CFSTR("MovieWindow"), &window );
    DisposeNibReference( nibRef );

    SetPortWindowPort( window );

    AdjustMovieWindow( gMovie, window );

    AEDisposeDesc( &newDescriptor );
}


Movie  GetMovieFromFile( FSSpec userFileFSSpec )
{
    OSErr     err;
    Movie     movie = NULL;
    short     movieRefNum;
    short     movieResID = 0;

    err = OpenMovieFile( &userFileFSSpec, &movieRefNum, fsRdPerm );

    err = NewMovieFromFile( &movie, movieRefNum, &movieResID,
                            NULL, newMovieActive, NULL );

    CloseMovieFile( movieRefNum );

    return ( movie );
}


void  AdjustMovieWindow( Movie movie, WindowRef window )
{
    Rect   movieBox;

    SetMovieGWorld( movie, NULL, NULL );

    GetMovieBox( movie, &movieBox );
    OffsetRect( &movieBox, -movieBox.left, -movieBox.top );
    SetMovieBox( movie, &movieBox );
```

```
    SizeWindow( window, movieBox.right, movieBox.bottom, TRUE );
    ShowWindow( window );

    GoToBeginningOfMovie( gMovie );
    MoviesTask( gMovie, 0 );
}


void  PlayOneMovie( Movie movie )
{
    GoToBeginningOfMovie( movie );

    StartMovie( movie );

    do
    {
        MoviesTask( movie, 0 );
    } while ( IsMovieDone( movie ) == FALSE );
}
```

For More Information

The following web sites provide extra information about some of this chapter's topics:

- **Navigation Services:**
  `http://developer.apple.com/techpubs/macosx/Carbon/Files/NavigationServices/navigationservices.html`
- **QuickTime technologies:**
  `http://developer.apple.com/techpubs/quicktime/quicktime.html`
- **QuickTime API:**
  `http://developer.apple.com/techpubs/quicktime/qtdevdocs/RM/frameset.htm`
- **Movie controllers:**
  `http://developer.apple.com/techpubs/quicktime/qtdevdocs/RM/frameset.htm`