QuickTime Kit Programming Guide

QuickTime > Cocoa

2005-11-09

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction to QuickTime Kit Programming Guide

The QuickTime Kit is a new framework (`QTKit.framework`) developed by Apple for working with QuickTime movies in Cocoa applications on Mac OS X. The QuickTime Kit framework, which offers a rich API for manipulating time-based media, is designed as an alternative to and eventual replacement for the existing Cocoa Application Kit classes NSMovie and NSMovieView. Using this new API provides developers with more extensive coverage of QuickTime functions and data types than is offered by those Application Kit classes and achieves this in a way that minimizes the requirement for Cocoa programmers to be conversant with Carbon data types such as handles, aliases, file-system specifications, and the like.

The QuickTime Kit also comes with a new QuickTime palette, which lets you drag a QuickTime movie object into your project window, and display, control, and edit that movie without writing a single line of code.

To work with this new framework, you don't need to know anything about the existing NSMovie and NSMovieView classes, but you should be familiar with developing Cocoa applications using Xcode and Interface Builder. Because the framework is flexible and relatively easy to use in Cocoa, you won't need to have a comprehensive understanding of the QuickTime C API in order to build your application or extend its functionality.

The QuickTime Kit framework is available in Mac OS X v10.4 and later. The framework also supports applications running in Mac OS X v10.3, but requires QuickTime 7 or later.

Who Should Read This Document

If you are a Cocoa developer who wants to integrate QuickTime movies in your application, you should read the material presented in this document. The various QuickTime and Cocoa mailing lists provide a useful developer forum for raising issues and answering questions that are posted.

If you are new to Cocoa or QuickTime, you should read these webpages, which are intended to get you up to speed with both Apple technologies: *Getting Started with Cocoa* and *Getting Started with QuickTime*.

Organization of This Document

This document follows a progressive, learn-as-you-go structure. Each chapter depends, to a certain extent, on understanding the material in any previous chapters.

- "The QuickTime Kit API" (page 11) describes the various classes and functions in the QuickTime Kit and some of their possible uses.
- "Building a Simple QTKitPlayer Application" (page 15) explains how you can build a simple QTKitPlayer application.
- "Extending the QTKitPlayer Application" (page 27) discusses how you can extend the functionality of the QTKitPlayer application by adding Cocoa code to your Xcode project.

- "Adding New Capabilities to the QTKitPlayer Application" (page 57) discusses how you can add a simple Cocoa drawer with a timer to your QTKitPlayer application and how you can take advantage of the QuickTime C API.
- "Extending the QTKitPlayer To Stream Audio and Video" (page 75) describes how you can add the streaming of audio and video to your QTKitPlayer application.
- "Adding Multimedia Playback Capability" (page 95) explains in detail how you can extend your QTKitPlayer application to add multimedia playback capabilities.

See Also

This introductory and tutorial document is designed as companion text to the reference material in *QTKit Framework Reference*. The various classes and methods in the QuickTime Kit framework are described in detail therein. Have that document handy as you learn the API and work through the various steps you need to follow in building a QuickTime application.

The QuickTime Kit API

The QuickTime Kit framework was developed by Apple to provide support for the most common media-related needs of Cocoa developers. This support was accomplished by using certain abstractions and data types familiar to Cocoa programmers and by defining other abstractions and data types that are new––but only where necessary. This chapter describes the various Objective-C classes and methods implemented with QuickTime that make up the QuickTime Kit framework and discusses some of their features and possible uses.

You'll want to read this chapter, which is brief, for an overview of the QuickTime Kit classes and functions. For more detailed information, refer to *QTKit Framework Reference*.

If you want to see the framework header files, you can find them in the Mac OS X `/System/Library/Frameworks` directory as `QTKit.framework`. The new QuickTime palette resides in the `/Developer/Extras/Palettes` directory as `QTKit.palette`.

In the next chapter, you'll work with the new QTKit palette and explore its capabilities in building a simple QTKitPlayer application.

Working With the QuickTime Kit API

The QuickTime Kit is an Objective-C API designed for the basic manipulation of media, including movie playback, editing, and import and export to standard media formats.

The framework is at once powerful, yet easy to use in your Cocoa application. The QTKit palette provided in Interface Builder, for example, lets you simply drag a QuickTime movie object, complete with a controller for playback, into a window, and then set attributes for the movie––all of this without writing a single line of code.

Notably, the QuickTime Kit provides Objective-C classes that are suitable for use within a wide range of Cocoa-based software, including applications with a GUI and tools intended to run in a "headless" environment. For example, you can use the QuickTime Kit framework to write command-line tools that manipulate QuickTime movie files.

One distinct advantage in working with the QuickTime Kit framework is that it does not require, in most cases, a thorough knowledge of the QuickTime C API, which can be in itself something of a daunting task. (The QuickTime C API contains over 2500 function calls and although those calls are documented, developers who are new to QuickTime may find the API more than a bit overwhelming.) Nor does it require an understanding of the fundamentals of Carbon, including but not limited to QuickDraw, the File Manager, and the Memory Manager.

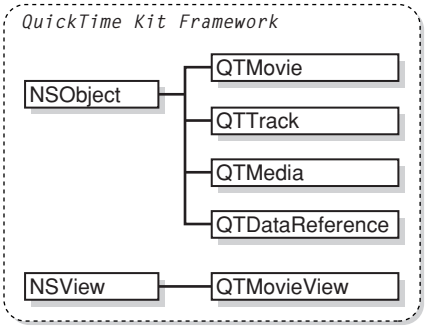
The QuickTime Kit classes are intended to replace NSMovie and NSMovieView, which will be deprecated in a future release of Mac OS X. If you are using those classes in your application, it is recommended that you plan to make a transition to the new QTMovie and QTMovieView classes in the QuickTime Kit framework. If you are writing an application designed to run in Mac OS X v10.4 and later, you should definitely move your code to QuickTime Kit in order to take advantage of this new functionality.

The classes in the current release of the QuickTime Kit framework are expected to grow in number and functionality as the framework evolves. This document reflects the current state of the art.

The QuickTime Kit Classes

The QuickTime Kit framework contains only five classes, along with a number of useful methods, functions, and protocols. Using these classes and methods, you can display, control, and edit QuickTime movies in your Cocoa application. Figure 1-1 shows the QuickTime Kit framework's class hierarchy.

Figure 1-1 The QuickTime Kit framework class hierarchy



Two of these new classes––QTMovie and QTMovieView––are similar in concept to the existing Application Kit classes NSMovie and NSMovieView, but they offer the advantage of providing greater functionality than those classes, which were limited to simple movie playback and rudimentary pasteboard editing. They also expose more of QuickTime's capabilities in an object-oriented fashion. For example, QuickTime Kit provides Cocoa methods for working with data references, thus reducing the need for developers to work with Macintosh Carbon APIs, such as the File Manager and the Memory Manager.

The principal class in the QTKit framework is QTMovie. This class serves as a Cocoa representation of a QuickTime movie and its associated movie controller. QTMovie objects typically can be associated with movie data contained in files, URLs, memory blocks, the pasteboard, or even an existing open QuickTime movie. QTMovie provides methods for getting and setting movie properties and for editing movies.

To display a QTMovie object in a window, you use the QTMovieView class. QTMovieView is a subclass of NSView that supports movie playback and editing. A movie controller bar is displayed in the view, but can optionally be hidden.

A QuickTime movie consists of one or more tracks, each of which is associated with a single media. Similarly, a QTMovie is associated with one or more objects of type QTTrack, each of which is associated with a single QTMedia object. The QTTrack and QTMedia classes provide a number of methods for operating on QuickTime tracks and media.

The QTDataReference class represents a data reference, which is QuickTime's standard way of picking out movie data, whether stored in files, URLs, memory blocks, or elsewhere.

The QuickTime Kit also defines the `QTTime` and `QTTimeRange` structures for representing specific times and time ranges in a movie or track.

QTMovie

The QTMovie class represents a QuickTime movie, which is a collection of playable and editable media content. A movie describes the sources and types of the media in that collection and their spatial and temporal organization. These collections may be used for presentation (such as playback on the screen) or for the organization of media for processing (such as composition and transcoding to a different compression type).

Just as a QuickTime movie contains a set of tracks, each of which defines the type, the segments, and the ordering of the media data it presents, a QTMovie object is associated with instances of the QTTrack class. In turn, a QTTrack object is associated with a single QTMedia object.

A QTMovie object can be initialized from a file, from a resource specified by a URL, from a block of memory, from a pasteboard, or from an existing QuickTime movie.

Once a QTMovie object has been initialized, it is used in combination with a QTMovieView for playback.

The QTMovie class includes an extensive number of instance and class methods that let you perform a wide range of operations on QuickTime movies. Some of these capabilities include:

- Creating and initializing a QTMovie object
- Getting a list of supported file types
- Setting movie properties and attributes
- Getting and setting selection times
- Getting movie tracks and movie images
- Storing movie data
- Controlling movie playback
- Editing and saving a movie
- Getting QTMovie primitives—that is, getting the movie or movie controller associated with a QTMovie object

The QTMovie class also includes a large number of constants you can use to specify movie attributes.

QTMovieView

QTMovieView, a subclass of NSView, can be used to display and control QuickTime movies. A QTMovieView is typically used in combination with a QTMovie object, which supplies the movie being displayed. A QTMovieView also supports editing operations on the movie.

The movie may be placed within an arbitrary bounding rectangle in the view's coordinate system, and the remainder of the view can be filled with a fill color. The movie controller, if visible, can also be placed within an arbitrary bounding rectangle in the view's coordinate system.

QTTrack

The QTTrack class represents a QuickTime track (of type `Track`). QTTrack objects are associated with QTMovie objects and support methods for getting and setting the track properties. If necessary, you can retrieve the track identifier associated with a QTTrack object by calling its `quickTimeTrack:` method.

QTMedia

The QTMedia class represents a QuickTime media (of type `Media`). QTMedia objects are associated with QTTrack objects and support methods for getting and setting the media properties. If necessary, you can retrieve the media identifier associated with a QTMedia object by calling its `quickTimeMedia:` method.

QTDataReference

A QTDataReference object is a representation of a QuickTime data reference, which is used to specify the location of a movie or its media data. You can create QTDataReference objects that refer to data stored in files accessed using filenames or URLs, or in memory accessed using handles, pointers, or NSData objects.

QuickTime Kit Functions

The QuickTime Kit framework provides a number of functions for working with `QTTime` and `QTTimeRange` structures.

QTTime

The `QTTime` structure defines the value and time scale of a time.

Functions are available for creating a `QTTime` structure, getting and setting times, comparing `QTTime` structures, adding and subtracting times, and getting a description.

QTTimeRange

The `QTTimeRange` structure defines a range of time. It's used, for instance, to specify the active segment of a movie or track.

Functions are available for creating a `QTTimeRange` structure, querying time ranges, creating unions and intersections of time ranges, and getting a description.

Building a Simple QTKitPlayer Application

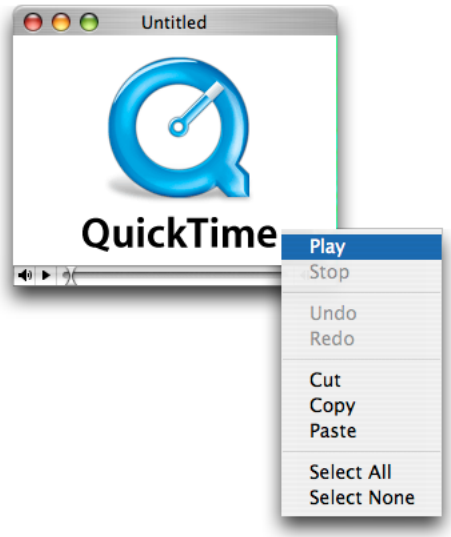
In this chapter, you'll build QTKitPlayer, a simple application that demonstrates some of the power and flexibility of the QuickTime Kit framework. When completed, your QTKitKPlayer application will allow you to open and play QuickTime movies, as shown in Figure 2-1. Amazingly, you won't have to write a single line of code to implement this media player.

Using Xcode as your integrated development environment (IDE), you'll see how easy it is to work with the QuickTime Kit framework. In this example, you'll use the new QTKit palette provided in the Interface Builder collection of palettes. The QTKit palette will do a lot of the work for you in constructing this application.

Figure 2-1 The completed QTKitPlayer application

Using the QTKit palette and Xcode, you'll be able to build a functioning media player application that displays and controls the playback of QuickTime movies, including QuickTime movies that support sprites, QuickTime VR, Flash, and 3GPP, among other file types. You'll even be able to add a contextual menu that includes editing commands, as shown in Figure 2-2––again, without writing a single line of code.

Figure 2-2 The completed QTKitPlayer application with a contextual menu



Creating the QTKitPlayer Project

If you've worked with Cocoa and Xcode before, you know that every Cocoa application starts out as a project. A project is simply a repository for all the elements that go into the application, such as source code files, frameworks, libraries, the application's user interface, sounds, and images. You use Xcode to create and manage your project.

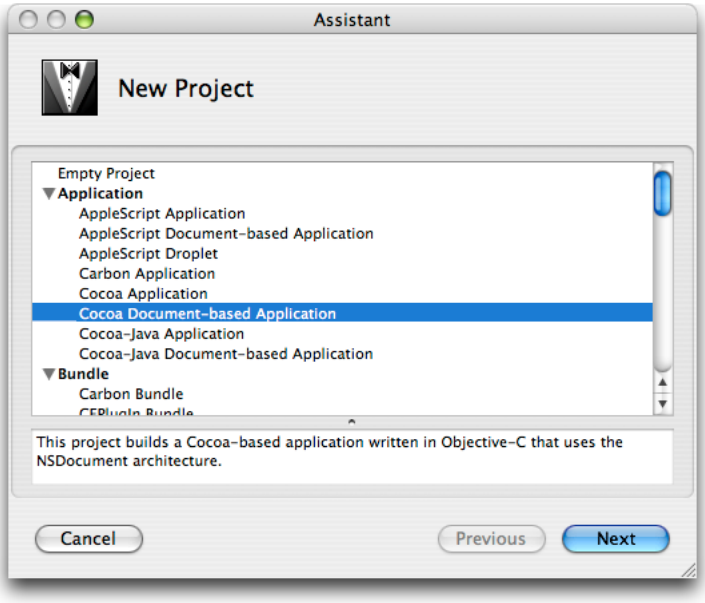
The QTKitPlayer application is a good learning example for developers who may be new to Cocoa and QuickTime. If you already know Cocoa, you probably won't be surprised at how quickly and effortlessly you can build this media player application.

To create the project, follow these steps:

1. Launch Xcode and choose File > New Project.

2. When the new project window appears, select Cocoa Document-based Application, as shown in Figure 2-3. Click Next.

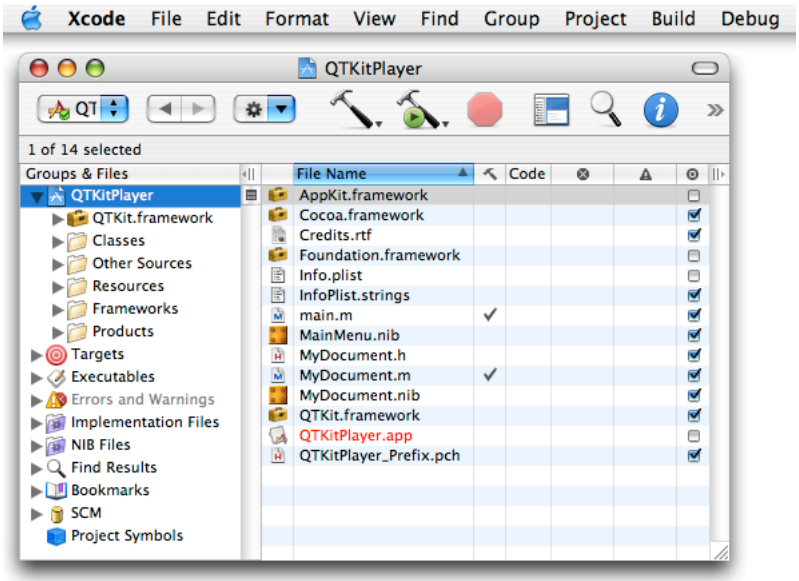
Figure 2-3 The Cocoa New Project Assistant window with the Cocoa Document-based Application option selected



3. Name the project `QTKitPlayer` and place it in the directory of your choice.

4. The Xcode project window appears as show in Figure 2-4. Verify the files shown are in your project. Note this illustration includes the QuickTime Kit framework, which you won't see until you add it in the next step.

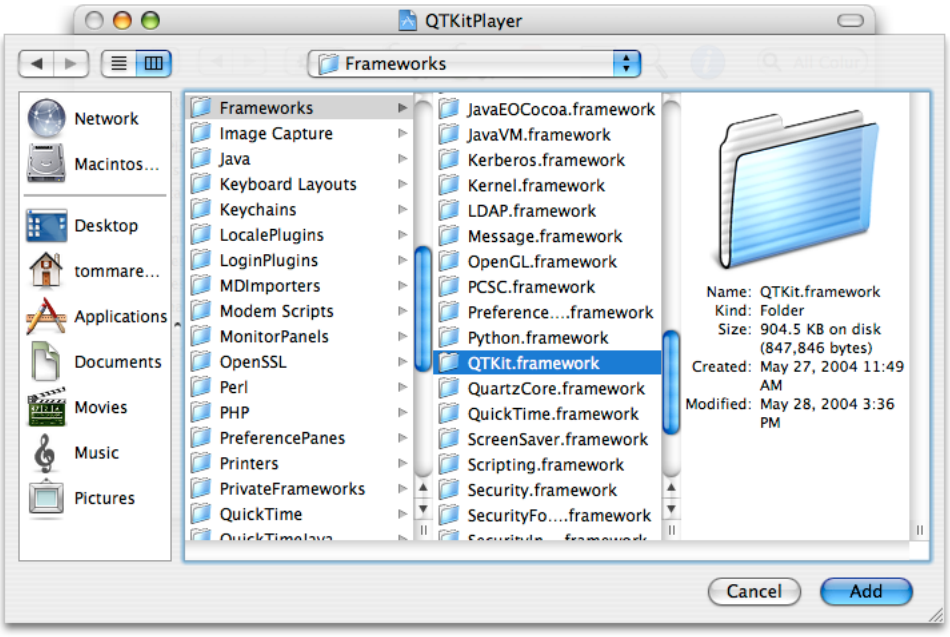
Figure 2-4 The QTKitPlayer application in Xcode



5. Next, you need to add the QuickTime Kit framework to your QTKitPlayer project. Choose Project > Add to Project.

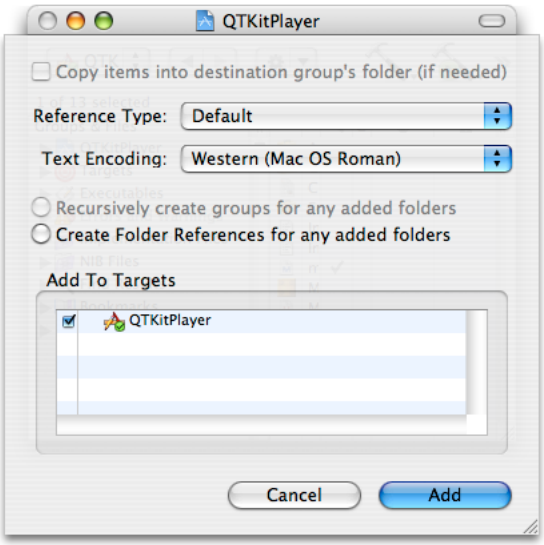
6. The QuickTime Kit framework resides in the `System/Library/Frameworks` directory. Select the framework, shown in Figure 2-5, and click Add to add it to your QTKitPlayer project.

Figure 2-5 The QuickTime Kit framework in the Frameworks directory



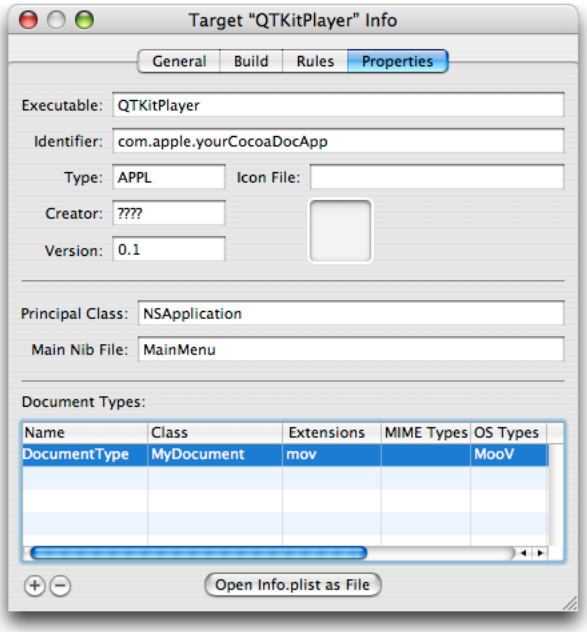
7. Another sheet opens, as shown in Figure 2-6. You don't need to change any of the settings, so just click Add to add the framework to the QTKitPlayer Target.

Figure 2-6 Adding QTKit.framework to the QTKitPlayer target



8. In your QTKitPlayer Xcode project window, open Targets in the Groups & Files list and then double-click QTKitPlayer. When the Target "QTKitPlayer" Info window opens, select the Properties pane, as shown in Figure 2-7.

Figure 2-7 The Target "QTKitPlayer" Info window with the Properties pane displayed



9. Select the DocumentType row and in the Extensions column, enter `mov` and in the OS Types column, enter `MooV`. Note that, strictly speaking, this step is not necessary if you use Interface Builder to specify the movie––it knows how to open movies.

The next section discusses how to use the new QTKit palette in Interface Builder to construct the QTKitPlayer application.

Working With the QTKit Palette

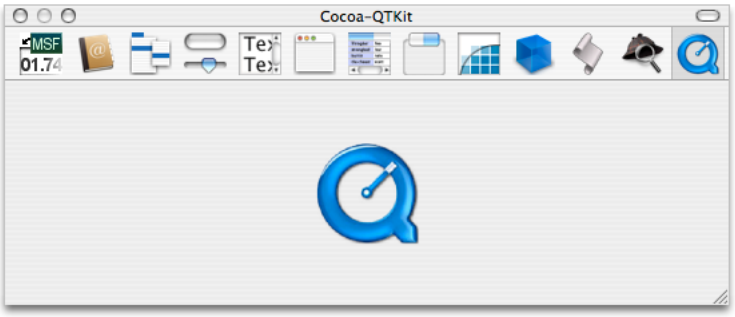
In this phase of constructing your QTKitPlayer application, you'll work with the new QTKit palette in Interface Builder. After you've completed these steps, you'll be able to build and run the QTKitPlayer in Xcode––without modifying any of your source files or adding any lines of code to them.

The QTKit palette resides in the `/Developer/Extras/Palettes` directory. To access the palette, you can either double-click to launch it in Interface Builder, or drag the palette into the `/Developer/Palettes` directory, in which case it will appear among the palettes in the toolbar shown in Figure 2-8. You can place the QTKit palette at any position in the row of palettes, in this case next to the Sherlock icon.

To finish building your QTKitPlayer, follow these steps:

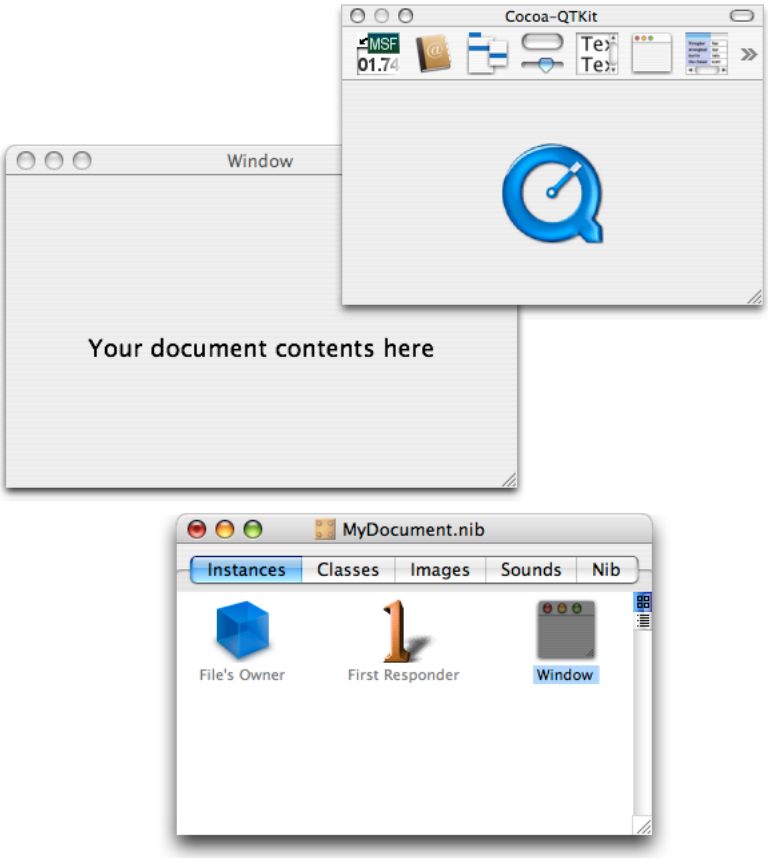
1. Open Interface Builder by double-clicking MyDocument.nib in your Xcode project window and select the blue QuickTime icon in the palettes collection at the far right, next to the Sherlock icon, as shown in Figure 2-8. Note that the QuickTime icon in this palette looks like the one available in the Cocoa-GraphicsViews palette. Be sure that you select the Cocoa-QTKit palette.

Figure 2-8 The Cocoa-QTKit palette



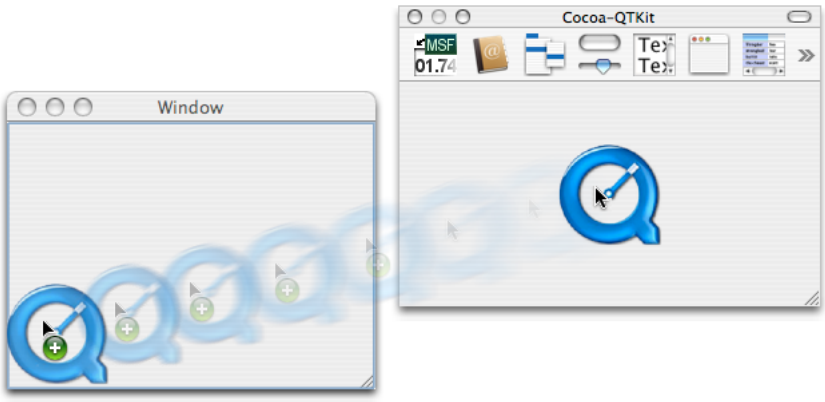
2. Delete the "Your document contents here" text in the window object shown in Figure 2-9.

Figure 2-9 MyDocument.nib with the QTKit palette and a window selected



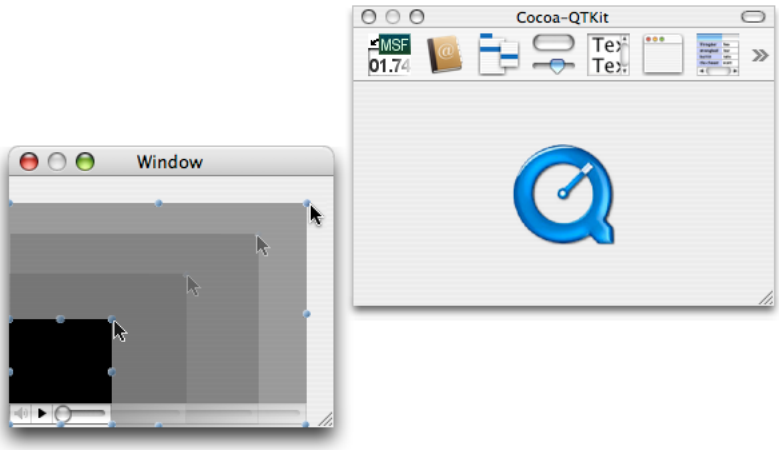
3. Drag the QTKit icon to the bottom-left corner of the project window, as shown in Figure 2-10.

Figure 2-10 The QTKit icon dragged to the application window



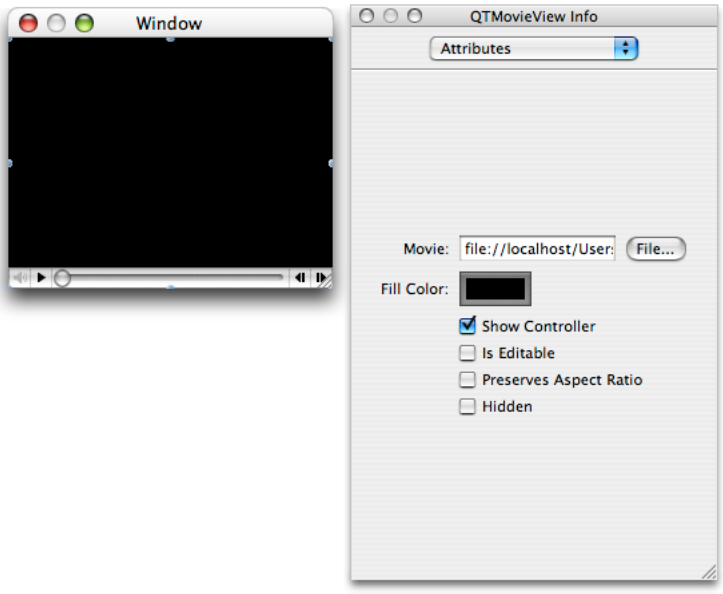
4. You now have a QuickTime movie view object with a control bar in the bottom-left corner of the window. Drag the QuickTime movie view object by its corner handle to the upper-right corner of the window, filling the entire window, so the movie view object with its control bar is visible, as shown in Figure 2-11.

Figure 2-11 The QuickTime movie view object enlarged to fill the entire contents of the window



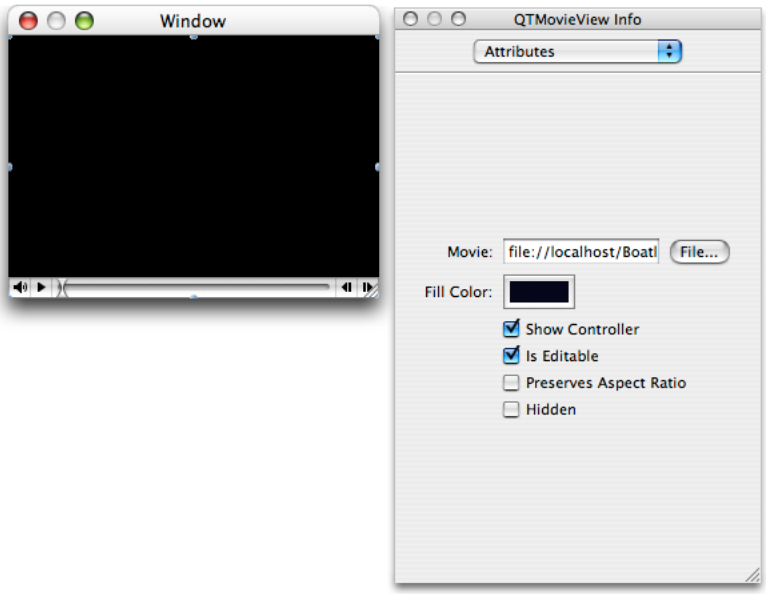
5. Click the QuickTime movie view object in the window, then press Command-1 to open the QTMovieView Info window. The Attributes pane is shown in Figure 2-12 with the default fill color of black and the Show Controller item selected. Note that the field that lets you select a movie from a list of files by clicking the File button is initially blank. The illustration in Figure 2-12 shows a partial file path for a movie to be selected.

Figure 2-12 The QTMovieView Attributes pane



6. To add editing capabilties to your QTKitPlayer, select the Is Editable item. Notice that the slider in the control bar in your player changes from a button to another shape, shown in Figure 2-13, similar to)(, which allows for editing of a movie.

Figure 2-13 The Is Editable item selected and the slider changed



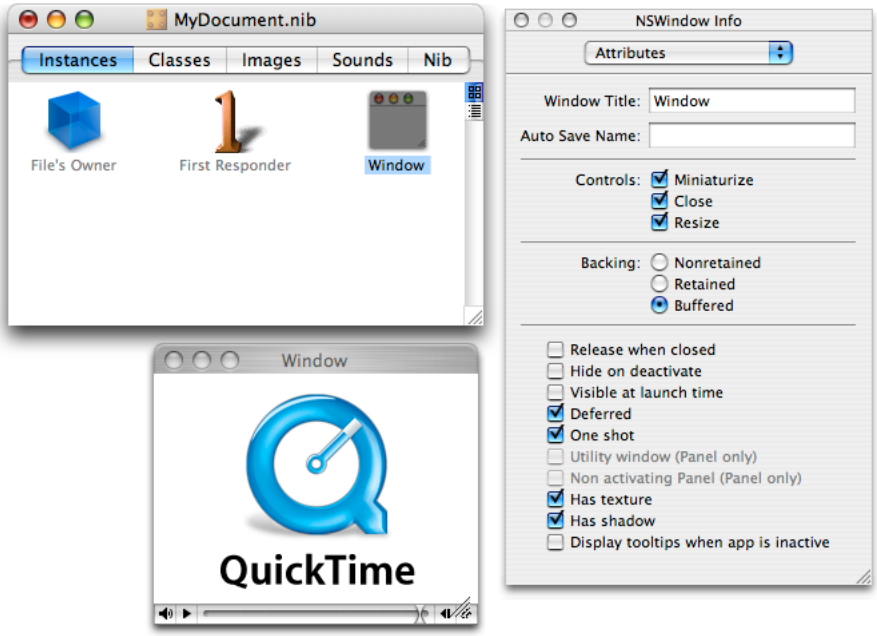
7. Now you're ready to specify a movie to play and edit. In the QTMovieView Info Attributes pane, click the File button.

8. In the open dialog that appears, select a QuickTime movie and click Open. Your player window will look similar to the one illustrated in Figure 2-14. Note that the movie in Figure 2-14 has movie editing enabled, as shown by the appearance of the slider in the control bar.

Figure 2-14 A QuickTime movie in your window

9. If you want to modify the attributes of the window in any way, you can open the NSWindow Info pane. To add texture to the movie window, for example, select the Window icon in the MyDocument.nib window and press Command-1 to open the NSWindow Info window with the Attributes pane displayed. Select the "Has texture" option, as shown in Figure 2-15.

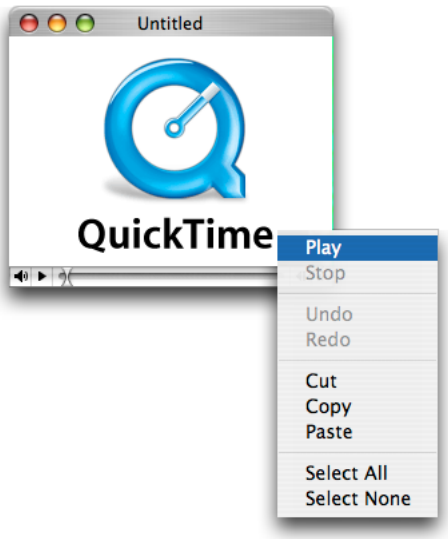
Figure 2-15 The QuickTime movie object window with texture added



10. Save the `MyDocument.nib` file and quit Interface Builder.

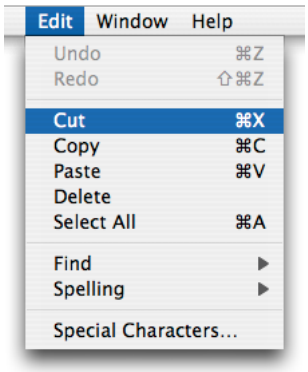
11. In Xcode, run and build the QTKitPlayer application. When the movie runs, it will open a document-based window with the movie you've specified as Untitled. The movie is completely editable. Control-click anywhere in the movie. A contextual menu appears, as shown in Figure 2-16.

Figure 2-16 The QTKitPlayer with an editable movie displayed



12. To access the editing features of the player, choose from the menu options shown in Figure 2-17 or use the contextual menu.

Figure 2-17 The Edit menu in QTKitPlayer



Extending the Functionality of QTKitPlayer

If you've followed correctly all the steps described in the previous section of this document, you'll have built successfully the QTKitPlayer application, which lets you display, control, and edit QuickTime movies. This is only the beginning, however. You can extend the functionality of your media player by taking advantage of the new classes and methods provided in the QuickTime Kit framework.

For example, with the addition of a few hundred lines of code, you can add import and export capabilities to your application, so it can play QuickTime VR movies, display still images, or export movies to different file formats, such as MPEG-4 and 3GPP.

If you're a Cocoa programmer, you'll already be able to envision the possibilities. If you're new to Cocoa or QuickTime, you'll be amazed how quickly and easily you can extend the functionality of your media player. The next chapter describes how you can accomplish this.

Extending the QTKitPlayer Application

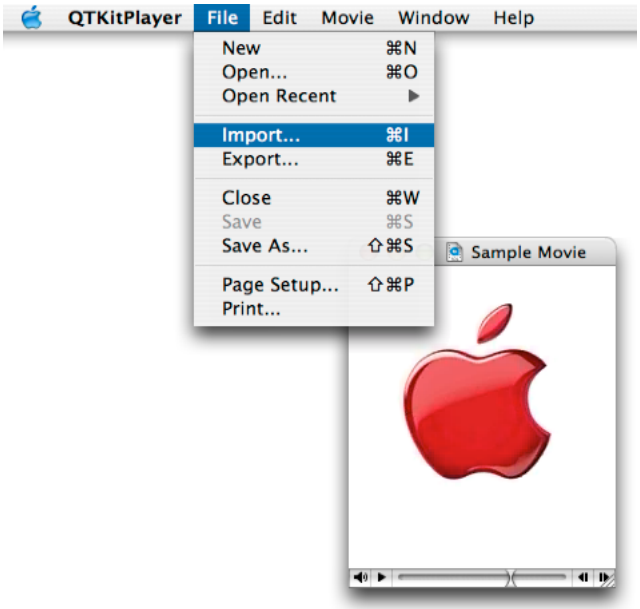
If you've built the simple QTKitPlayer application described in the previous chapter, you are now ready to extend its functionality by following the steps outlined in this chapter. If you skipped the previous chapter, you may want to familiarize yourself with the section "Working With the QTKit Palette" (page 19), which introduces the new Cocoa-QTKit palette available in the QuickTime Kit framework. This chapter assumes that, at very least, you have:

- A basic knowledge of Cocoa development as well as the fundamentals of the Objective-C programming language
- An understanding of Xcode, Interface Builder, and the QTKit palette

When completed, your extended QTKitKPlayer application will allow you not only to open and play QuickTime movies, but also to import and export them, as shown in Figure 3-1.

This added capability means that your application will be able to import and display a wide range of media types that QuickTime understands and supports in Mac OS X, including but not restricted to MP4 clips, JPEG images, and audio files, as you can see in the illustration in Figure 3-2 (page 28). In so doing, your application moves beyond a simple media player and provides useful functionality for handling a variety of media types and playback tasks.

Figure 3-1 The completed QTKitPlayer application with extended import and export capabilities



You'll be able to accomplish all of this by working with Interface Builder to construct the QTKitPlayer user interface and then adding several hundred lines of code to your Xcode project.

Note that the QTKitPlayer application runs in both Mac OS X v10.4 and Mac OS X v10.3. You need QuickTime Player 7 (the latest version) installed on your system, however, to take full advantage of the new QuickTime Kit framework capabilities.

Important: The complete source code for the extended QTKitPlayer application is available for Mac OS X v10.4 in the folder Developer > Examples > QuickTime > QTKit > QTKitPlayer. You may find it useful to build and compile the QTKitPlayer project in Xcode after you have worked through the steps in this programming guide. The QuickTime Kit Reference, which documents all the classes and methods in the QTKit framework, is also available in Developer > Examples > QuickTime > QTKit > QTKitObjC.pdf. The reference document is periodically updated at http://developer.apple.com/referencelibrary/QuickTime/index.html as new methods and classes are added.

Figure 3-2 The completed QTKitPlayer application with an MP4 movie, an MP3 audio clip, and a JPEG still image displayed

Beyond importing and exporting of QuickTime-supported media types, your completed QTKitPlayer application will also include editing capabilities more extensive than just cut, copy, and paste. You'll add editing features such as add, add scaled, replace, and trim, all of which are supported by the QTKit framework in Mac OSX 10.4. These features will enable you to edit QuickTime movies with greater control and precision, as described in Table 3-1. If you look at the illustration shown in Figure 3-3, you'll see the Edit menu in the QTKitPlayer with the additional commands available.

Table 3-1 Edit commands added to the QTKitPlayer application

Edit Command	Description
Add	Adds a new chunk to a movie in one or more new tracks––on top of what's already there––instead of inserting it as the Paste command does.
Add Scaled	Scales the chunk being pasted in time, and the chunk becomes stretched or compressed so that it plays for the same duration as the current selection in the receiving movie, which can result in slow-motion or fast-motion effects.

Edit Command	Description
Replace	Replaces the current selection in the receiving movie. If there is no current selection, the command will replace the entire movie.
Trim	Deletes everything in the current movie except the current selection.

Figure 3-3 The added editing capabilities in your QTKitPlayer project

Following the steps in this chapter, you'll also be able to provide more extensive control of movie playback and display, as shown in the Movie menu commands (Figure 3-4), including looping and cloning of your QuickTime movies.

Figure 3-4 The Movie menu commands for movie control and playback

Creating the Extended QTKitPlayer Project

Before you get started with your extended QTKitPlayer project, be sure that you are running either Mac OS X v10.4 or Mac OS X v10.3 and have the following items installed on your system:

- Xcode 2.0 or later and Interface Builder 2.5 or later. Note that you can use Xcode 1.1 or Xcode 1.5 to build your project, but to take full advantage of the new programming and navigational features available, such as class model visualization, you'll want to use Xcode 2.0.
- The QuickTime Kit framework, which is in the Mac OS X `/System/Library/Frameworks` directory as `QTKit.framework`
- The new QuickTime palette, which resides in the `/Developer/Extras/Palettes` directory as `QTKit.palette`. If you are running Mac OS X v10.4, the palette is installed automatically. If you are running Mac OS X v10.3, you will have to install it manually into the `/Developer/Palettes` directory, so that it appears in the Interface Builder palettes toolbar when you start up Interface Builder.
- QuickTime Player version 7

Note: To create the QTKitPlayer project, you can either develop on Mac OS X v10.4 or Mac OSX v10.3. If you're using Xcode 2.0 or later as your IDE, you'll want to develop your project on Mac OSX 10.4.

Now that you've verified you have all these items, let's get cooking.

Getting Started

You'll want to set aside the QTKitPlayer application you built following the steps in the previous chapter and start fresh with a new Xcode project.

There are a number of reasons for this, but suffice to say, you'll find it easier to extend your media player application by starting from scratch. You'll still be using the new QTKit palette to drag a QTMovieView object into a window in Interface Builder. The steps you take beyond that will nonetheless be different from those described in the previous chapter.

Note again that the QTKit palette resides in the `/Developer/Extras/Palettes` directory. To access the palette, you can either double-click its icon to launch it in Interface Builder, or drag the palette into the `/Developer/Palettes` directory, in which case it will appear among the palettes in the toolbar shown in Figure 3-6 (page 32) when Interface Builder starts up.

To create the project, follow these steps:

1. Launch Xcode and choose File > New Project.
2. When the new project window appears, select Cocoa Document-based Application. Click Next.
3. Name the project `QTKitPlayer` and place it in the directory of your choice. Note that you're naming the project the same as the project you built in the previous chapter, so you should place it in another directory.
4. Next, you need to add the QuickTime Kit framework to your QTKitPlayer project. Choose Project > Add to Project.

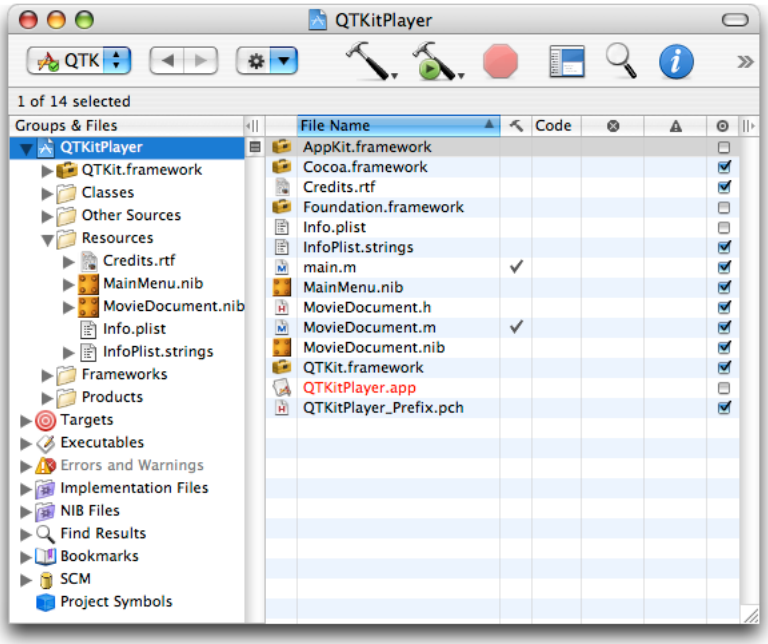
5. The QTKit framework resides in the `System/Library/Frameworks/QTKit.framework` directory. Select the framework, and click Add to add it to your QTKitPlayer project.

6. In the Add To Targets sheet, click the Add button.

7. Now you need to rename the nib, declaration, and implementation files with the Rename command in the File menu. Select each file in turn and rename `MyDocument.h` to `MovieDocument.h`, and `MyDocument.m` to `MovieDocument.m`. To rename the nib file, you need to open the `MyDocument.nib` in Interface Builder and save it as `MovieDocument.nib`, then delete the `MyDocument.nib`.Using Xcode 2.1, you won't be able to rename the nib from the File > Rename menu item, which is grayed out when you click a nib in the Project window.

8. The Xcode project window appears as shown in Figure 3-5, with your files renamed accordingly. Verify that the files shown in the illustration are the same as those in your project. Note that you won't have to rename the `MainMenu.nib` file.

Figure 3-5 The QTKitPlayer project in Xcode with nib, declaration, and implementation files renamed

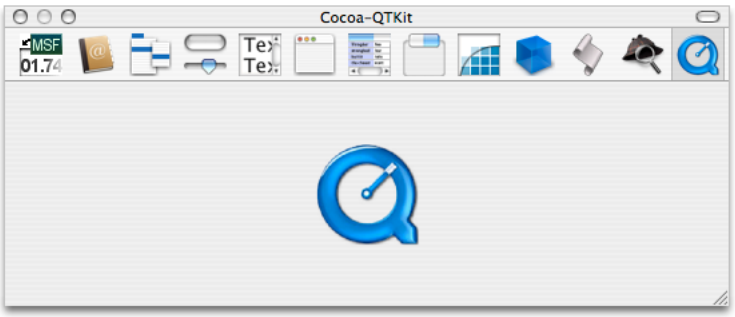


Creating the QTMovieView Object with Outlets and Actions

In this next sequence of steps, you'll work with the QTKit palette and add outlets and actions in Interface Builder to your Xcode project.

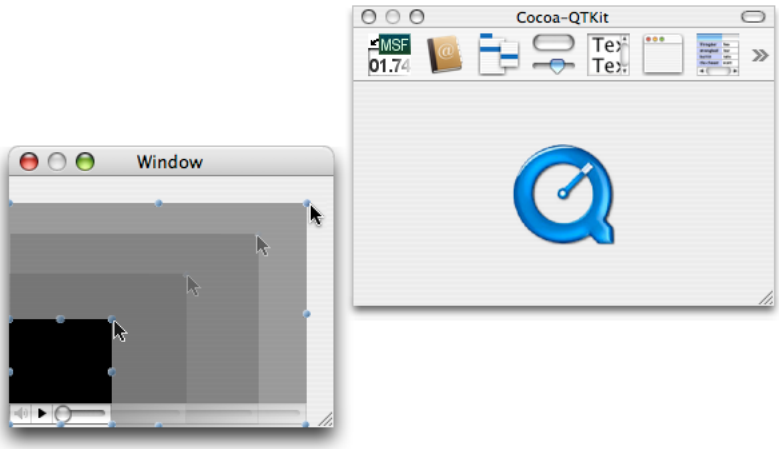
1. Open Interface Builder by double-clicking `MovieDocument.nib` in your Xcode project window and select the blue QuickTime icon in the toolbar, as shown in Figure 3-6. Note that the QuickTime icon in this palette looks like the one available in the Cocoa-GraphicsViews palette, but they are different. The one in the QTKit palette gives you the QTMovieView object rather than an NSMovieView object. Be sure that you select the item in the Cocoa-QTKit palette.

Figure 3-6 The Cocoa-QTKit palette



2. Delete the text "Your document contents here" in the window object.
3. Drag the QuickTime icon from the QTKit palette to the bottom-left corner of the project window.
4. You now have a QuickTime movie view object with a control bar in the bottom-left corner of the window. Drag the upper-right handle of the movie view object to the upper-right corner of the window, filling the entire window, so the movie view object with its control bar is visible, as shown in Figure 3-7.

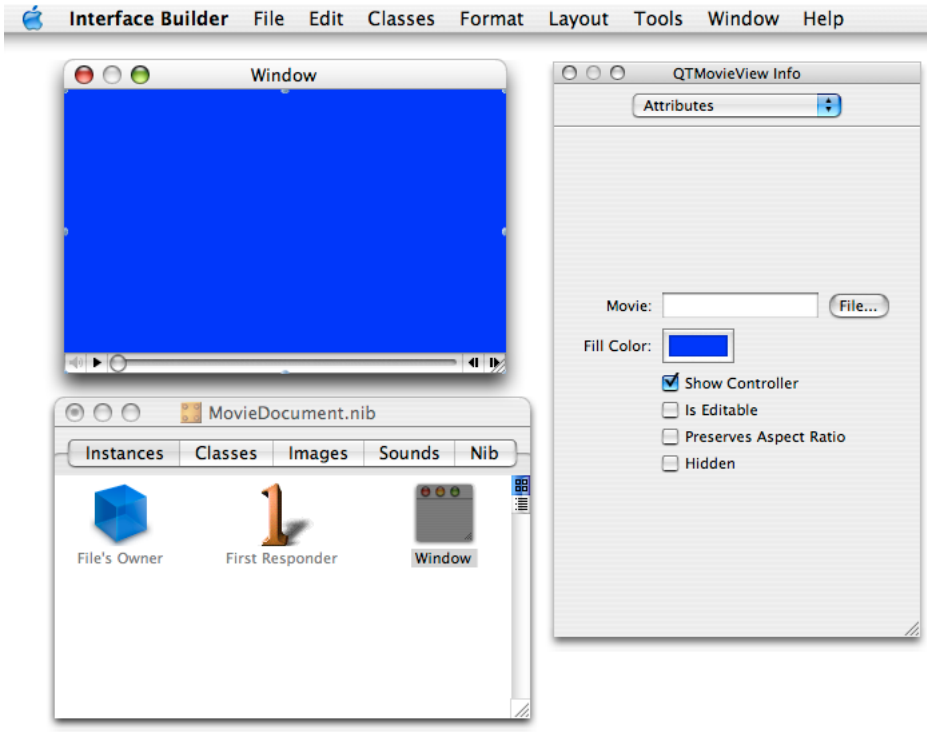
Figure 3-7 The QuickTime movie view object dragged to fill the entire contents of the window



5. Click the QuickTime movie object in the window, then press Command-1 to open the QTMovieView Info window. The Attributes pane appears with the default fill color of black and the Show Controller item selected. Note that the field that lets you select a movie from a list of files by clicking the File button is initially blank. Don't click the File button to open a QuickTime movie that will be displayed in your window.
6. Leave the Show Controller item selected.

7. If you want to change the fill color of the movie view, click the Fill Color box and choose a new color. Figure 3-8 shows the window with a fill color of blue.

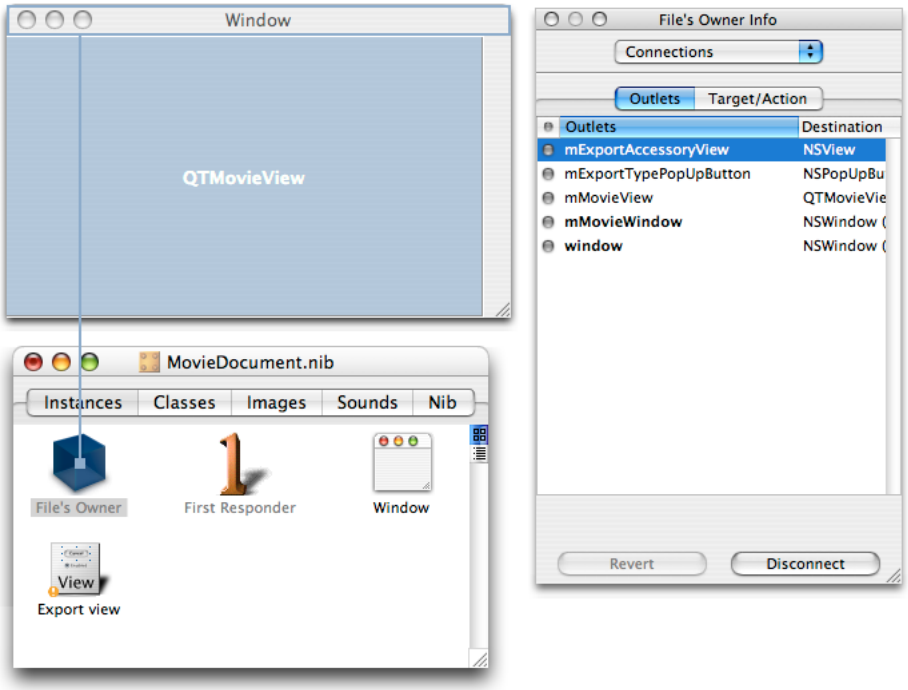
Figure 3-8 The QTMovieView Info pane with a blue fill color selected for the QTMovieView object window



8. Next, you want to add outlets to your QTKitPlayer project. Double-click File's Owner in your `MovieDocument.nib` window.

9. Enter `mMovieView` and `QTMovieView`, and `mMovieWindow` and `NSWindow` in their respective fields of the Connections pane, as shown in Figure 3-9. The illustration shows the Export view object in the `MovieDocument.nib` window, as well as the outlet `mExportAccessoryView` and its destination as `NSView`, and the outlet `mExportTypePopUpButton` and its destination as `NSPopUpButton`. You'll enter these in a later step.

Figure 3-9 The File's Owner connections to various outlets, with actions specified



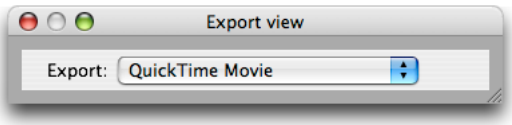
10. Select the File's Owner icon and press the Control key. A wire appears with a small square at one end. Make sure the square lands on the File Owner's icon. You want to hook up the File's Owner to the QTMovieView object. Select the outlet `mMovieView` and click Connect to wire up the File's Owner with the QTMovieView object, as shown in Figure 3-9.

11. Repeat the same step to connect the File's Owner icon to the NSWindow object. Choose the outlet `mMovieWindow` and click Connect to wire up the File's Owner with the NSWindow object.

Creating the Export View Object

In this sequence of steps, you want to create an Export view object in Interface Builder, as shown in Figure 3-10. This will enable your application to display a dialog and pop-up menu when you want to export a QuickTime movie to another file type.

Figure 3-10 The export view pop-up menu constructed in Interface Builder

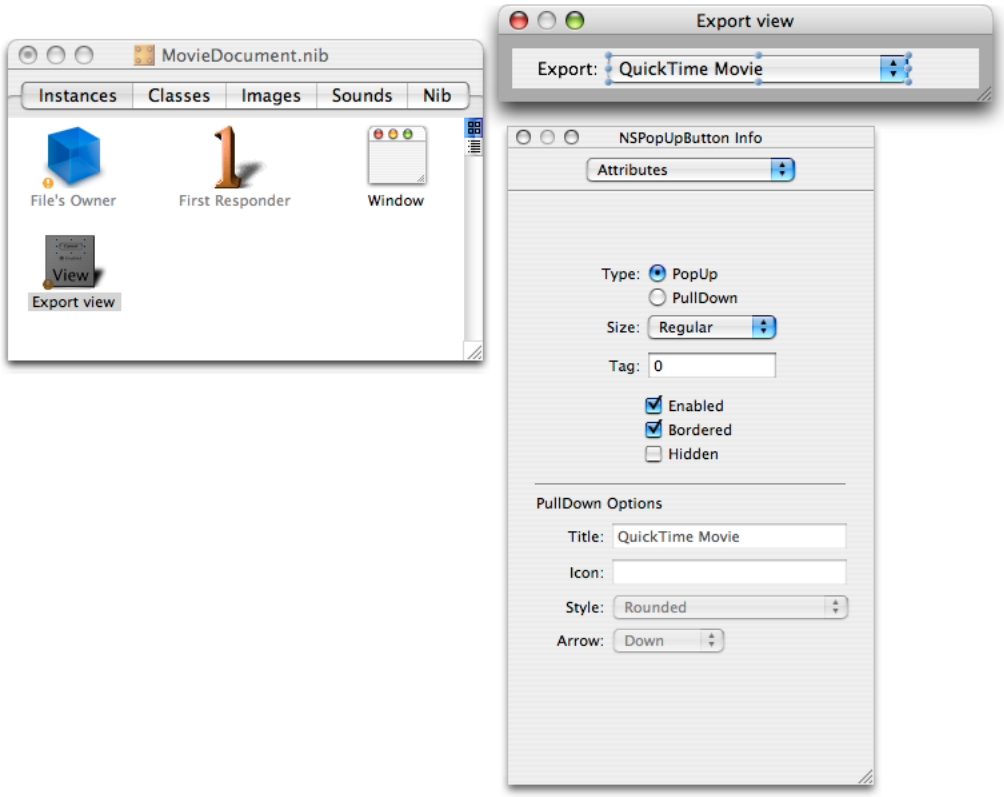


1. In the Interface Builder palette toolbar, choose Cocoa-Containers and drag the CustomView container to your `MovieDocument.nib` window. It appears as a View object.

2. Rename the View object `Export view`.

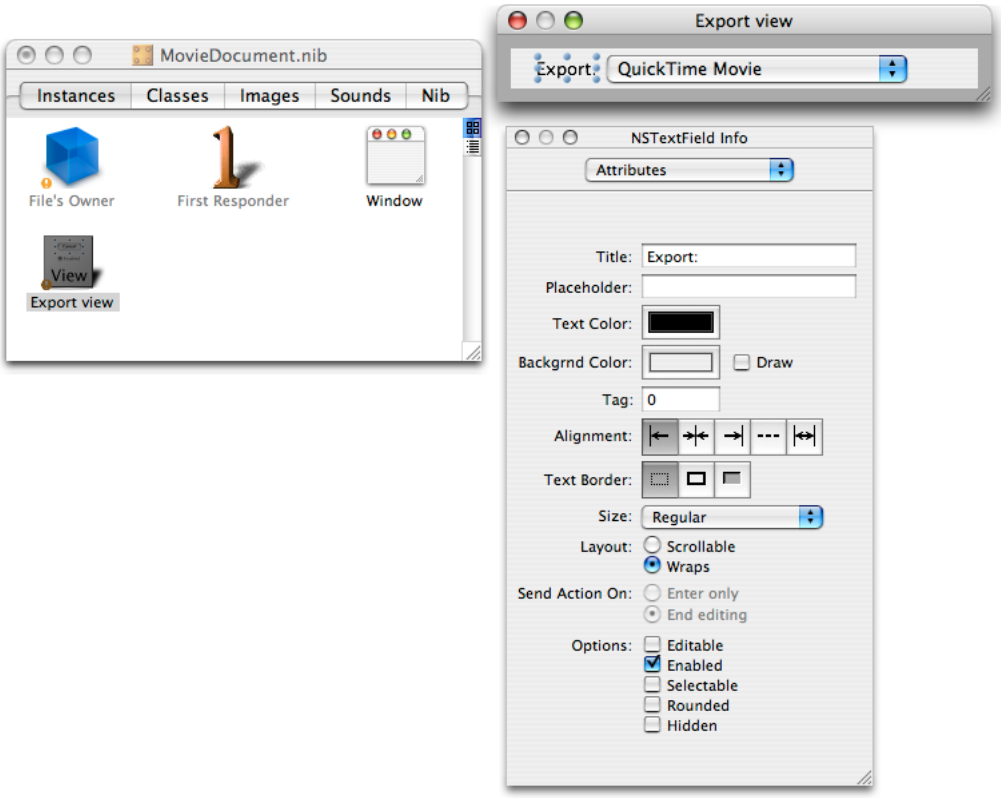
3. Now you want to add a pop-up menu labeled Export to the Export view object. Drag a pop-up menu from the Cocoa-Controls palette into the Export view window. In the Attributes pane of the NSPopUpButton Info window, type `QuickTime Movie` in the Title field, as shown in Figure 3-11.

Figure 3-11 The export view object with the NSPopUpButton Info window displayed and various attributes specified



4. To add a label for the pop-up menu, drag a text field from the Cocoa-Text palette and position it to the left of the pop-up menu as shown in Figure 3-12. In the Attributes pane of the NSTextField Info window, type `Export:` in the Title field. In the Options section, deselect the Editable option and the Selectable option.

Figure 3-12 The export view object with the NSTextField Info window displayed and various attributes specified



5. Now you are ready to wire up the Export view object. Select the File's Owner icon and press the Control key. You want to hook up the File's Owner to the Export view object. Choose the outlet `exportAccessoryView` and click Connect to wire up the File's Owner with the NSView object. Select the outlet `exportTypePopUpButton` and click Connect to wire up the File's Owner with the NSPopUpButton object.

Adding Outlets and Actions to The MovieView Object

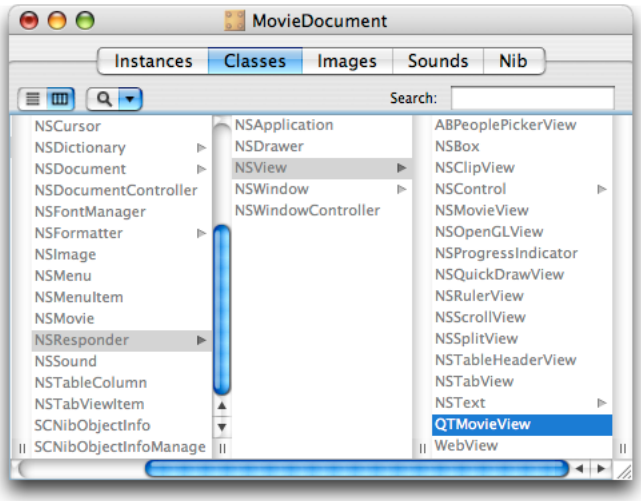
Now you want to add outlets and actions to the MovieView object.

1. In the MovieDocument.nib window, double-click the File's Owner icon. The MovieDocument Class Info window Attributes pane appears with four outlets selected and one deselected.

2. To add actions to the MovieDocument class, click the Actions tab, click Add, and type `doClone:`. Repeat for the following actions: `doExport:`, `doLoop:`, `doLoopPalindrome:`, `doPreservesAspectRatio:`, `doSetFillColor:`, `doSetFillColorPanel:`, `doSetPosterTime:`, and `doShowController:`. These are some of the actions that you want to add to your QTKitPlayer through various menu items.

3. Now you want to add more actions to your QTMovieView object using Interface Builder. In the MovieDocument.nib window, click the Classes tab, scroll down the list, and select the NSResponder class. From the list of subclasses, select NSView. Scroll down again and select QTMovieView, as shown in Figure 3-13.

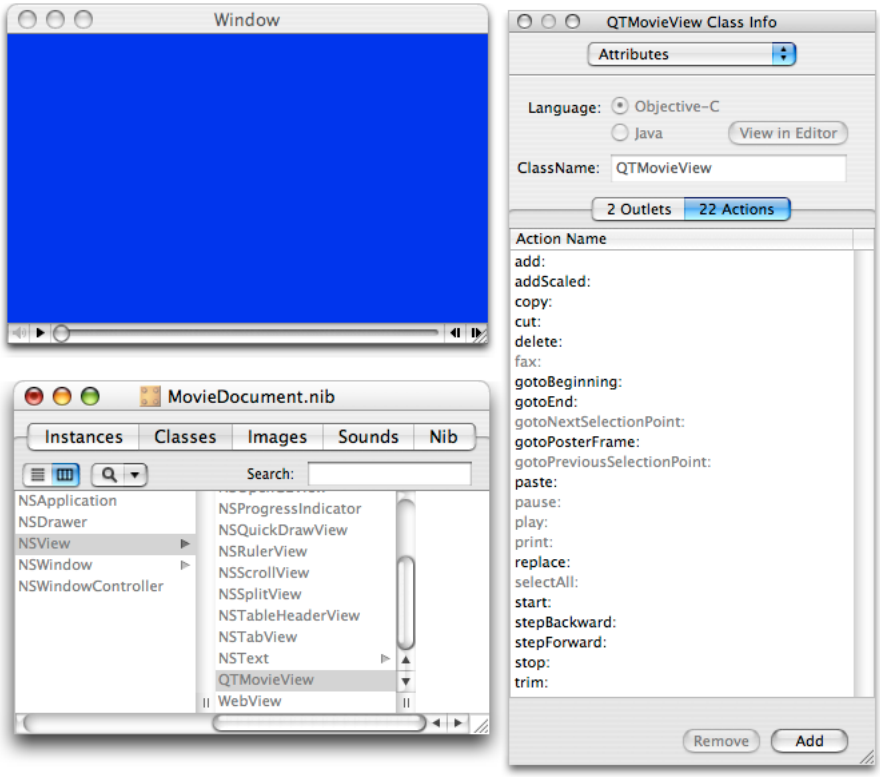
Figure 3-13 The QTMovieView subclass selected in MovieDocument nib file



4. Press Command-1 to open the QTMovieView Class Info window. This displays the attributes of your QTMovieView object, deselected in grayed out text. You want to add these actions: `add:`, `addScaled:`, `copy:`, `cut:`, `delete:`, `gotoBeginning:`, `gotoEnd:`, `gotoPosterFrame:`, `paste:`, `replace:`, `stepBackward:`, `stepForward:`, and `trim:`.

To add these actions, you need to parse the QTMovieView header file. In Interface Builder, select Classes > Read Files... A dialog opens. Select System > Library > Frameworks > QTKit.framework > Headers > QTMovieView.h. Click Parse. The QTMovieView window now shows the actions hilighted and added(Figure 3-14).

Figure 3-14 The QTMovieView Class Attributes pane with list of actions added to the QTMovieView object

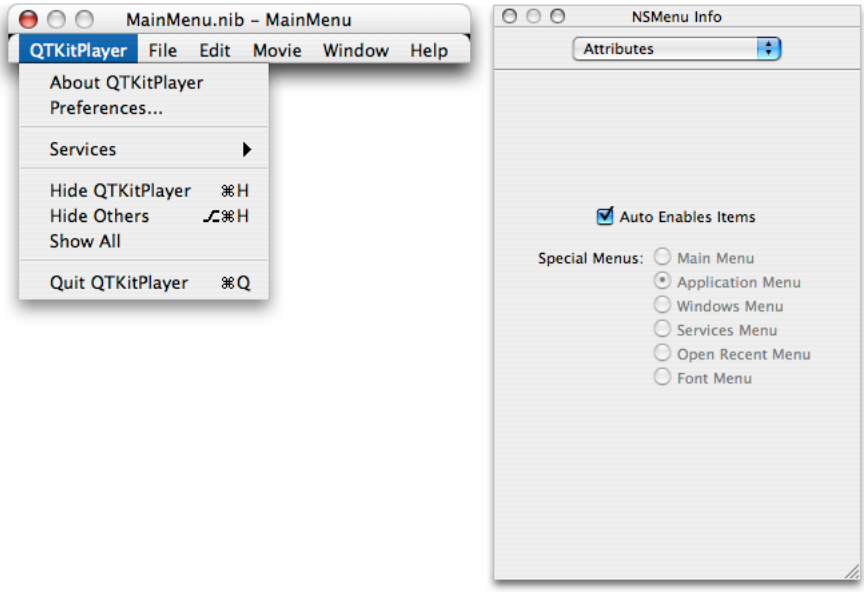


Wiring Up The MainMenu.nib

In this next sequence of steps, you want to change the NSApplication menu title in the MainMenu.nib - MainMenu file to QTKitPlayer and add items to each of the menu choices. You'll also want to modify the attributes of each NSMenuItem.

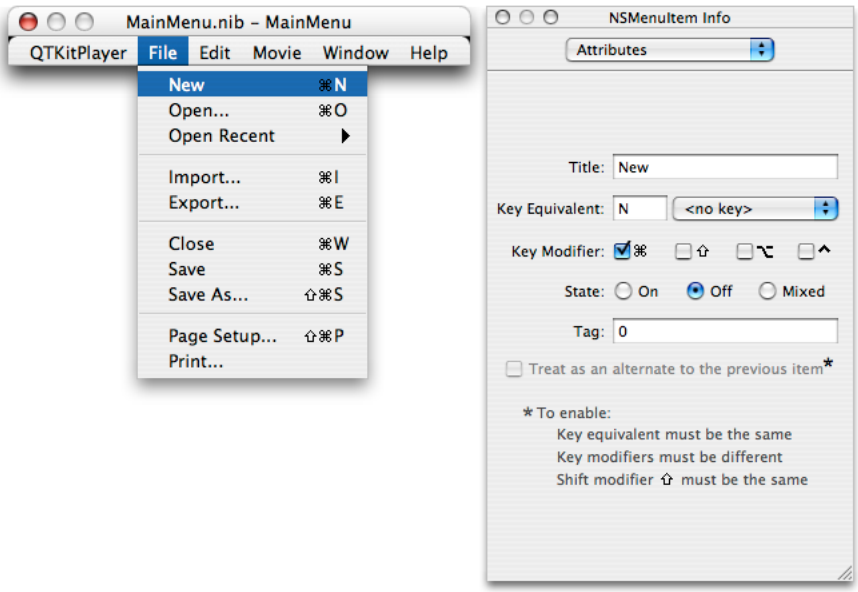
1. In the MainMenu.nib - MainMenu file, you want to change the NSApplication menu title to QTKitPlayer. Select the menu title so that the item is highlighted. Then, double-click the item and enter the new text `QTKitPlayer`. Do this for the About, Hide, and Quit menu items in the MainMenu.nib window, so that each item refers to the QTKitPlayer, as shown in Figure 3-15.

Figure 3-15 The MainMenu nib file with changes to the application menu



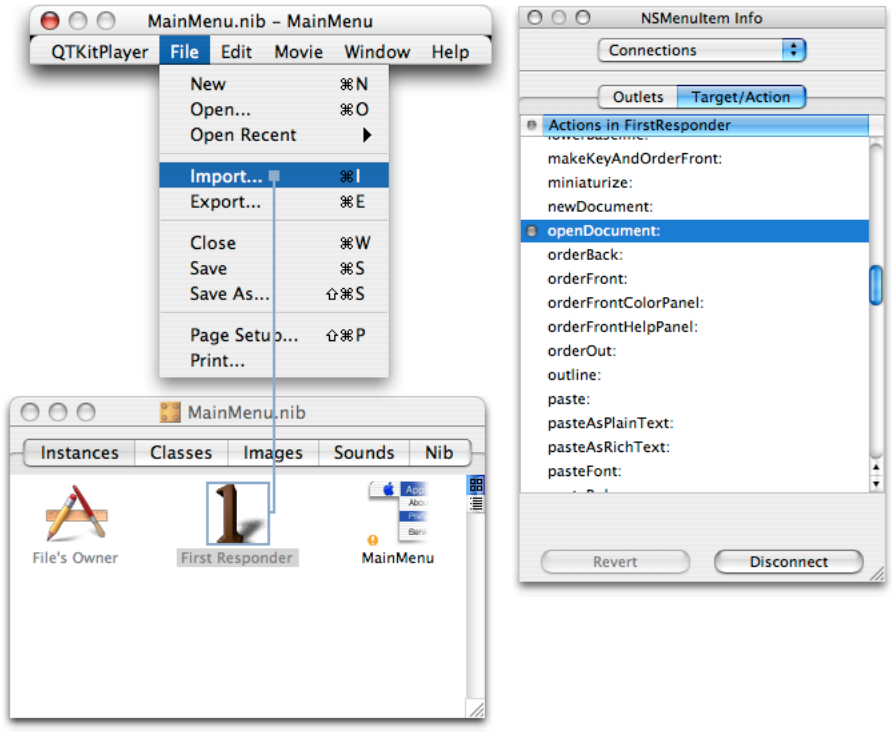
2. In the MainMenu.nib - MainMenu file, add to the File menu the items Import and Export, so your File menu appears as shown in Figure 3-16.

Figure 3-16 The MainMenu nib file additions to the File menu



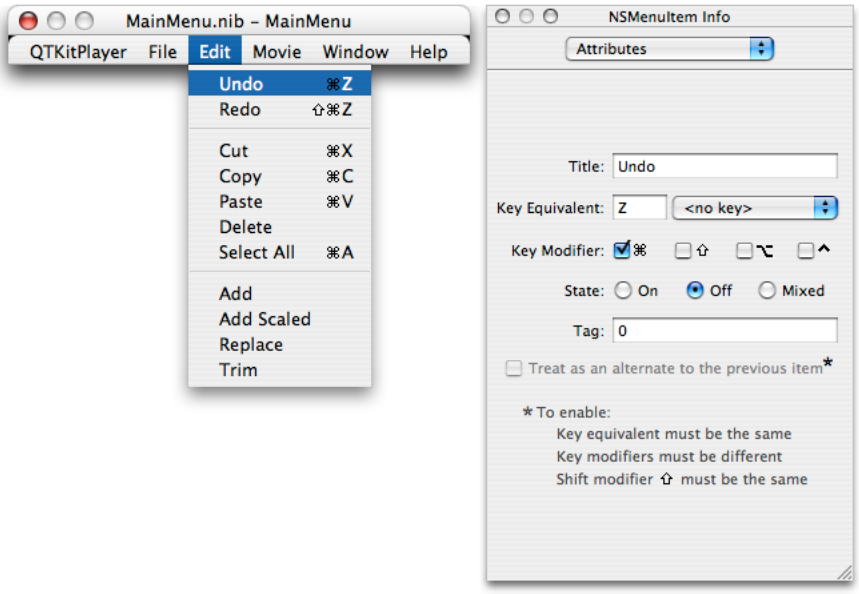
3. In the File menu, select the Import menu item and press Command-2 to open the Connections pane of the NSMenuItem Info window. Press Control and hook up the Import menu item to First Responder. Scroll down the list of the actions for First Responder and select openDocument, as shown in Figure 3-17. Click the Connect button.

Figure 3-17 The MainMenu nib file with the Import menu item hooked up to First Responder and the openDocument action connected



4. In the MainMenu.nib - MainMenu file, add to the Edit menu the items shown in Figure 3-18. These items include Undo, Redo, Cut, Copy, Paste, Delete, Select All, Add, Add Scaled, Replace, and Trim.

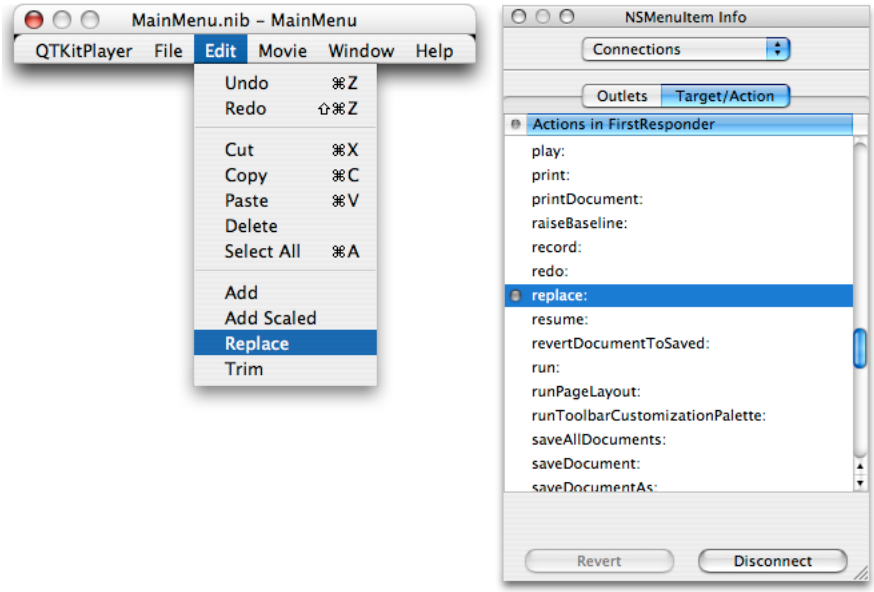
Figure 3-18 The MainMenu nib file with additions to the Edit menu



5. Before you hook up these items to actions in First Responder, you need to double-click the First Responder icon in the MainMenu.nib. The info panel brings up a list of the attributes available. Select the Actions tab. Now you want to add the following actions: `add:`, `addScaled:`, `copy:`,`cut:`, `delete:`, `gotoBeginning:`, `gotoEnd:`, `gotoPosterFrame:`, `paste:`, `replace:`, `stepBackward:`, `stepForward:`, and `trim:` Enter each action and click Add.

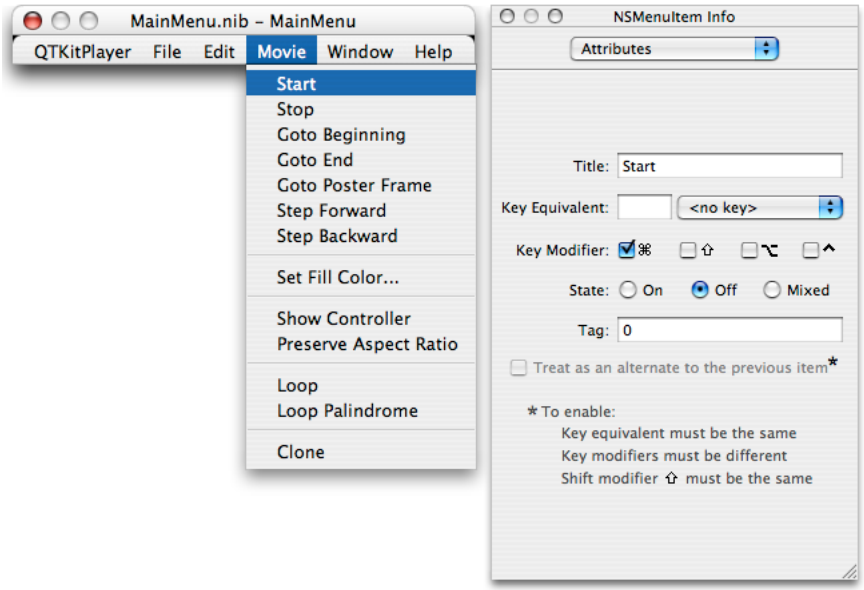
Now you want to hook up these items to actions in First Responder. In the Edit menu, select each of the new menu items and connect it to the appropriate action in the actions list for First Responder, as shown in Figure 3-19.

Figure 3-19 The MainMenu nib file with the Replace menu item selected and its connection specified



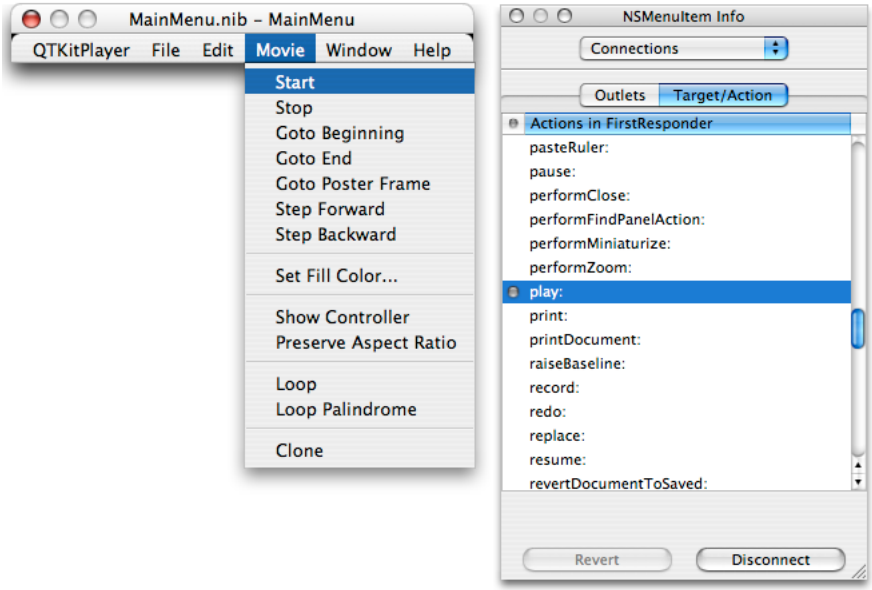
6. In the MainMenu.nib - MainMenu file, add the Movie menu and those items shown in Figure 3-20.

Figure 3-20 The MainMenu nib file with additions to the Movie menu



7. Hook up these items to actions in First Responder. In the Movie menu, select each item and connect it to the appropriate action. Rename Start to `play:` and Stop to `pause:`, as shown in Figure 3-21.

Figure 3-21 The MainMenu nib file with the Start menu item selected and its connection specified



8. Save your `MainMenu.nib` file and quit Interface Builder.

You've now completed the first part of your project. In the next part, you'll add the Cocoa code that makes your QTKitPlayer application work and perform like a champion.

Adding Code to the MovieDocument Class Interface

In this next sequence of steps, you'll be adding a small amount of code to your `MovieDocument.h` class interface file.

To begin, open the `MovieDocument.h` declaration file in your Xcode project and delete any existing code in the file. Now follow these steps:

1. Insert the following import code at the beginning of your file:

```
#import <Cocoa/Cocoa.h>
#import <QTKit/QTKit.h>
```

2. Add the following declaration code after your import statements:

```
@interface MovieDocument : NSDocument
{
 // movie window
    IBOutlet NSWindow *mMovieWindow;
    IBOutlet QTMovieView *mMovieView;
```

```
    // export
        IBOutlet NSView *mExportAccessoryView;
        IBOutlet NSPopUpButton *mExportTypePopUpButton;

    // movie document
        QTMovie         *mMovie;
}
```

The first line defines the MovieDocument class and specifies that it inherits from the NSDocument class. The first four statements declare the outlets you set up and connected in Interface Builder. The `movieWindow` instance variable, for example, points to the NSWindow object while the `movieView` instance variable points to the QTMovieView object. The last line declares that the `mMovie` instance variable points to the QTMovie object.

3. Define a class method with the following line of code:

```
+ (id)movieDocumentWithMovie:(QTMovie *)movie;
```

4. To initialize movies, add this code line:

```
- (id)initWithMovie:(QTMovie *)movie;
```

5. For the NSMenu validation protocol, which returns a `BOOL` value, as it validates the menu items in your project, add this:

```
- (BOOL)validateMenuItem:(NSMenuItem *)menuItem;
```

6. The following line of code, which enables you to set a movie to the QTMovie object using the `setMovie:` method, is one of the most common you'll use in building projects with the QuickTime Kit framework. Add it:

```
- (void)setMovie:(QTMovie *)movie;
```

7. Insert the following block of action method declarations in your `MovieDocument.h` file:

```
- (IBAction)doExport:(id)sender; // run the export sheet
- (IBAction)doSetFillColorPanel:(id)sender; // update the fill color
- (IBAction)doSetFillColor:(id)sender;
- (IBAction)doSetPosterTime:(id)sender;
- (IBAction)doShowController:(id)sender;//toggle controller visibility
- (IBAction)doPreserveAspectRatio:(id)sender;// toggle aspect ratio
- (IBAction)doLoop:(id)sender; // toggle looping
- (IBAction)doLoopPalindrome:(id)sender; // toggle palindrome looping
- (IBAction)doClone:(id)sender;
```

8. To handle NSDocument overrides, add the following lines of code:

```
- (NSString *)windowNibName;
- (void)windowControllerDidLoadNib:(NSWindowController*)windowController;
- (NSData *)dataRepresentationOfType:(NSString *)docType;
- (BOOL)writeWithBackupToFile:(NSString *)fullDocumentPath ofType:(NSString
*)type saveOperation:(NSSaveOperationType)saveOperation;
- (BOOL)readFromFile:(NSString *)fileName ofType:(NSString *)type;
- (void)printShowingPrintPanel:(BOOL)showPanels;
```

9. Insert the following method before the `@end` directive in the file:

```
- (BOOL)createMovieDocumentWithFile:(NSString *)fileName asData:(BOOL)asData;
```

```
@end
```

10. To save your NSSavePanel delegate, which handles your export dialog, insert these lines:

```
- (void)exportPanelDidEnd:(NSSavePanel *)sheet returnCode:(int)returnCode
contextInfo:(void *)contextInfo;
```

11. Save your `MovieDocument.h` file (Command-S).

This completes the first part of code additions to your Xcode project.

Adding Code to MovieDocument.m

In this next sequence of steps, you'll be adding a larger chunk of code to your `MovieDocument.m` implementation file.

To begin, open the `MovieDocument.m` file in your Xcode project and delete any existing code in the file. Now follow these steps:

1. Insert the following import code at the beginning of your file:

```
#import "MovieDocument.h"
#import <QTKit/QTKit.h>
```

2. Following your import statements, you want to define a constant for movies that are not necessarily visual––that is, audio files. Specify the width as 320. Insert these lines:

```
#define  kDefaultWidthForNonvisualMovies  320
@implementation MovieDocument
```

3. Now you want to add the following class method and deal with memory allocation. Note that this is just a convenience method if you want to create a document given a movie. By calling `autorelease`, you auto release the movie. Add these lines:

```
+ (id)movieDocumentWithMovie:(QTMovie *)movie
{
    return [[(MovieDocument *)[self alloc] initWithMovie:movie] autorelease];
}
```

4. Next, you need to add initialization code to initialize a MovieDocument instance and set the file type. Setting the movie to `nil` will release it. Add these lines:

```
- (id)initWithMovie:(QTMovie *)movie
{
    [super init];

    // init
    [self setFileType:@"MovieDocument"];
    [self setMovie:movie];

    if (mMovie == nil)
    {
        [self release];
        self = nil;
```

```
    }
    return self;
}
```

5. Insert the following code to handle deallocation of memory and notifications:

```
- (void)dealloc
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
    [self setMovie:nil];
    [super dealloc];
}
```

6. To deal with NSMenu validations for enabling and disabling of menu items, insert this chunk of code:

```
- (BOOL)validateMenuItem:(NSMenuItem *)menuItem
{
    BOOL valid = NO;
    SEL action;

    // init
    action = [menuItem action];
    // validate
    if (action == @selector(doExport:))
        valid = (mMovie != nil);

    else if (action == @selector(doSetFillColorPanel:))
        valid = YES;

    else if (action == @selector(doShowController:))
    {
        [menuItem setState:([mMovieView isControllerVisible] ? NSOnState :
NSOffState)];
        valid = YES;
    }
    else if (action == @selector(doPreserveAspectRatio:))
    {
        [menuItem setState:([mMovieView preservesAspectRatio] ? NSOnState :
NSOffState)];
        valid = (mMovie != nil);
    }
    else if ((action == @selector(doLoop:)) && mMovie)
    {
        [menuItem setState:([[mMovie attributeForKey:QTMovieLoopsAttribute]
boolValue] ? NSOnState : NSOffState)];
        valid = YES;
    }
    else if ((action == @selector(doLoopPalindrome:)) && mMovie)
    {
        if ([[mMovie attributeForKey:QTMovieLoopsAttribute] boolValue])
        {
            [menuItem setState:([[mMovie
attributeForKey:QTMovieLoopsBackAndForthAttribute] boolValue] ? NSOnState :
NSOffState)];
            valid = YES;
        }
    }
```

```
    else if (action == @selector(doClone:))
        valid = (mMovie != nil);

    else
        valid = [super validateMenuItem:menuItem];

    return valid;
}
```

7. To set the instance variable of the movie (one of the most common operations in the QuickTime Kit framework), add the following code:

```
- (void)setMovie:(QTMovie *)movie
{
    [movie retain];
    [mMovie release];
     mMovie = movie;
}
```

8. The next block of code is a bit more complicated. You want to set the movie window size to fit the movie, so that it is the "natural" movie size. (Natural size is the size of the movie when it was first created. After opening the movie, you can resize it.) The code here sets the coordinates, specifying that the movie window appears in the top-left corner of the window. Otherwise, the movie appears in the bottom-left corner. The code lets you move the movie around, but pegs it initially to the top-left corner. Insert the following code:

```
- (void)sizeWindowToMovie:(NSNotification *)notification
{
    NSRect   currWindowBounds, newWindowBounds;
    NSPoint  topLeft;
    static BOOL nowSizing = NO;

    if (nowSizing)
        return;

    nowSizing = YES;

    NSSize contentSize = [[mMovie attributeForKey:QTMovieCurrentSizeAttribute]
 sizeValue];
    if ([mMovieView isControllerVisible])
        contentSize.height += [mMovieView controllerBarHeight];

// get the current location and size of the movie window, so we can keep // the
 top-left corner pegged, i.e. fixed
    currWindowBounds = [[mMovieView window] frame];
    topLeft.x = currWindowBounds.origin.x;
    topLeft.y = currWindowBounds.origin.y + currWindowBounds.size.height;

    if (contentSize.width == 0)
        contentSize.width = currWindowBounds.size.width;

   newWindowBounds = [[mMovieView window] frameRectForContentRect:NSMakeRect(0,
 0, contentSize.width, contentSize.height)];

    [[mMovieView window] setFrame:NSMakeRect(topLeft.x, topLeft.y -
newWindowBounds.size.height, newWindowBounds.size.width,
newWindowBounds.size.height) display:YES];
```

```
    nowSizing = NO;
}
```

9. To deal with NSDocument overrides, add these lines:

```
- (NSString *)windowNibName
{
    return @"MovieDocument";
}
```

10. To set up the movie view and set the editable state of the controller, you need to add this chunk of code. You also want to "listen" for any movie size changes, so notifications are added here. Insert the following:

```
- (void)windowControllerDidLoadNib:(NSWindowController *)windowController
{

    // set up the movie view
    if (mMovie)
    {
       NSSize contentSize = [[mMovie attributeForKey:QTMovieNaturalSizeAttribute]
 sizeValue];
        [mMovieView setMovie:mMovie];
        if ([mMovieView isControllerVisible])
        {
          contentSize.height += [mMovieView movieControllerBounds].size.height;
            if (contentSize.width == 0)
            {
                contentSize.width = kDefaultWidthForNonvisualMovies;
            }
        }
        [[mMovieView window] setContentSize:contentSize];
    }
    else
        [mMovieView setMovie:[QTMovie movie]];

    // set the editable state
    [mMovie setAttribute:[NSNumber numberWithBool:YES]
forKey:QTMovieEditableAttribute];

    // watch for movie resizings
    if (mMovie)
    [[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(sizeWindowToMovie:) name:QTMovieSizeDidChangeNotification
object:mMovie];
}
```

11. To write to handle updates and saves, add the following code:

```
- (BOOL)writeWithBackupToFile:(NSString *)fullDocumentPath ofType:(NSString
*)type saveOperation:(NSSaveOperationType)saveOperation
{
    BOOL success = NO;

    // update/save
    if (saveOperation == NSSaveOperation)
        success = [mMovie updateMovieFile];
    else
```

```
        success = [super writeWithBackupToFile:fullDocumentPath ofType:type
saveOperation:saveOperation];

    return success;
}
```

12. Add these lines of code:

```
- (NSData *)dataRepresentationOfType:(NSString *)docType
{
    return [mMovie movieFormatRepresentation];
}
```

13. Add these lines to handle reading the movie from a file:

```
- (BOOL)readFromFile:(NSString *)fileName ofType:(NSString *)type
{
    BOOL  success = NO;
    NSData  *data;

    // read the movie
    if ([QTMovie canInitWithFile:fileName])
    {
        if ([type isEqualTo:@"MovieDocumentData"])
        {
            data = [NSData dataWithContentsOfFile:fileName];

        }
        else
        {
            [self setMovie:((QTMovie *)[QTMovie movieWithFile:fileName])];
            success = (mMovie != nil);
        }
    }

    return success;
}
```

14. The following chunk of code is intended to deal with opening movies and other media that QuickTime "understands" using an instance method and setting a flag. Add these lines:

```
- (id)openDocumentWithContentsOfFile:(NSString *)fileName
display:(BOOL)displayFlag
{
    // first check whether we already have a document for this file
    NSDocumentController *sharedDocController = [NSDocumentController
sharedDocumentController];
    MovieDocument *movieDocument = [sharedDocController
documentForFileName:fileName];

    if (movieDocument)
    {// we've got one, just bring it forward
        if (displayFlag)
        {
            [movieDocument showWindows];
        }
    }
    else
```

```
    {// we must create a new document.
        // we always create a movie document regardless of type
        // init
        movieDocument = [[[MovieDocument allocWithZone:[self zone]]
initWithContentsOfFile:fileName ofType:@"mov"] autorelease];

        // set up the document
        if (movieDocument)
        {

            // add the document
            [sharedDocController addDocument:movieDocument];

            // set up the document
            if ([sharedDocController shouldCreateUI])
            {
                [movieDocument makeWindowControllers];
                if (displayFlag)
                {
                    [movieDocument showWindows];
                }
            }
        }
    }
    return movieDocument;
}
```

15. Add the following code to locate the file to be opened:

```
- (BOOL)panel:(id)sender shouldShowFilename:(NSString *)filename
{
    BOOL isDir = NO;

   [[NSFileManager defaultManager] fileExistsAtPath:filename isDirectory:&isDir];

    if (isDir)
    {
        return YES;
    }
    else
    {
        return [QTMovie canInitWithFile:filename];
    }
}
```

16. Add this code to filter files through an Open dialog and display them:

```
- (void)openDocument:(id)sender {
    NSOpenPanel *openPanel = [NSOpenPanel openPanel];

    [openPanel setDelegate:self];
    [openPanel setAllowsMultipleSelection:NO];

    // files are filtered through the panel:shouldShowFilename: method above
    if ([openPanel runModalForTypes:nil] == NSOKButton) {
       [self openDocumentWithContentsOfFile:[[openPanel filenames] lastObject]
 display:YES];
    }
```

```
}
```

17. Add the following lines of code:

```
- (BOOL)application:(NSApplication *)theApplication openFile:(NSString *)filename
{
    return (nil != [self openDocumentWithContentsOfFile:filename display:YES]);
}
```

18. Insert the following code to display the Print dialog:

```
- (void)printShowingPrintPanel:(BOOL)showPanels
{
    NSPrintOperation *printOperation;

    // init
    printOperation = [NSPrintOperation printOperationWithView:mMovieView
printInfo:[self printInfo]];
    [printOperation setShowPanels:showPanels];

    // print
    [self runModalPrintOperation:printOperation delegate:nil didRunSelector:nil
 contextInfo:nil];
}
```

19. You add the following code to handle the NSSavePanel delegate for the export sheet. Note that you use an NSArray to list export types, such as `mpg4`, `bmpf`, `aiff`, and so on. These types are hard-coded here. You can add to the list if you want to and include, for example, `3gpp` or `mpg3`.

```
- (void)exportPanelDidEnd:(NSSavePanel *)sheet returnCode:(int)returnCode
contextInfo:(void *)contextInfo
{
    int             selectedItem;
    NSMutableDictionary*settings = nil;
    static NSArray *exportTypes = nil;

    // init
    if (exportTypes == nil)
    {
        exportTypes = [[NSArray arrayWithObjects:
            [NSNumber numberWithLong:'AIFF'], [NSNumber numberWithLong:'BMPf'],
 [NSNumber numberWithLong:'dvc!'],
            [NSNumber numberWithLong:'FLC '], [NSNumber numberWithLong:'mpg4'],
 [NSNumber numberWithLong:'MooV'],
             [NSNumber numberWithLong:'embd'], nil] retain];
    }

    // export
    if (returnCode == NSOKButton)
    {
        // init
        selectedItem = [mExportTypePopUpButton indexOfSelectedItem];
        settings = [NSMutableDictionary dictionaryWithCapacity:5];
       [settings setObject:[NSNumber numberWithBool:YES] forKey:QTMovieExport];

        if ((selectedItem >= 0) && (selectedItem < [exportTypes count]))
            [settings setObject:[exportTypes objectAtIndex:selectedItem]
forKey:QTMovieExportType];
```

```
        // export
        if (![mMovie writeToFile:[sheet filename] withAttributes:settings])
            NSRunAlertPanel(@"Error", @"Error exporting movie.", nil, nil, nil);
    }
}
```

20. Add the following actions to your file:

```
- (IBAction)doExport:(id)sender
{
    NSSavePanel *savePanel;

    // init
    savePanel = [NSSavePanel savePanel];

    // run the export sheet
    [savePanel setAccessoryView:mExportAccessoryView];
   [savePanel beginSheetForDirectory:nil file:[[self fileName] lastPathComponent]
 modalForWindow:mMovieWindow modalDelegate:self
        didEndSelector:@selector(exportPanelDidEnd: returnCode: contextInfo:)
contextInfo:nil];
}

- (IBAction)doSetFillColorPanel:(id)sender
{
    NSColorPanel *colorPanel;

    // init
    colorPanel = [NSColorPanel sharedColorPanel];
    [colorPanel setAction:@selector(doSetFillColor:)];
    [colorPanel setTarget:self];
    [colorPanel setColor:[mMovieView fillColor]];

    // run the panel
    [colorPanel makeKeyAndOrderFront:nil];
}

- (IBAction)doSetFillColor:(id)sender
{
    // update the fill color
    [mMovieView setFillColor:[sender color]];
}

- (IBAction)doShowController:(id)sender
{
    // toggle the controller visibility
    [mMovieView setControllerVisible:([sender state] == NSOffState)];
}

- (IBAction)doPreserveAspectRatio:(id)sender
{
    // toggle the aspect ratio preservation
    [mMovieView setPreservesAspectRatio:([sender state] == NSOffState)];
}

- (IBAction)doLoop:(id)sender
{
```

```
    // toggle looping
    [mMovie setAttribute:[NSNumber numberWithBool:([sender state] == NSOffState)]
 forKey:QTMovieLoopsAttribute];
}

- (IBAction)doLoopPalindrome:(id)sender
{
    // toggle palindrome looping
    [mMovie setAttribute:[NSNumber numberWithBool:([sender state] == NSOffState)]
 forKey:QTMovieLoopsBackAndForthAttribute];
}

- (IBAction)doClone:(id)sender
{
    MovieDocument *movieDocument;

    // init
    movieDocument = [MovieDocument movieDocumentWithMovie:[[mMovie copy]
autorelease]];

    // set up the document
    if (movieDocument)
    {
        // add the document
        [[NSDocumentController sharedDocumentController]
addDocument:movieDocument];

        // set up the document
        [movieDocument makeWindowControllers];
        [movieDocument showWindows];
    }
}

@end
```

21. Save your `MovieDocument.m` file (Command-S).

This completes the steps for adding code to your `MovieDocument.m` implementation file. There is only one more step, described in the next section, before you can run and build your QTKitPlayer application.

Modifying the Info.plist File

The is the last step to complete before you can build and run your QTKitPlayer application.

1. In your Xcode project, double-click the `Info.plist` file to open it.

2. Delete the existing contents of the `Info.plist` file.

3. Replace the contents of the file with the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
```

```
<plist version="1.0">
<dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>English</string>
    <key>CFBundleDocumentTypes</key>
    <array>
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>mov</string>
            </array>
            <key>CFBundleTypeName</key>
            <string>MovieDocument</string>
            <key>CFBundleTypeOSTypes</key>
            <array>
                <string>MooV</string>
            </array>
            <key>CFBundleTypeRole</key>
            <string>Editor</string>
            <key>NSDocumentClass</key>
            <string>MovieDocument</string>
        </dict>
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>mov</string>
            </array>
            <key>CFBundleTypeName</key>
            <string>MovieDocumentData</string>
            <key>CFBundleTypeOSTypes</key>
            <array>
                <string>????</string>
            </array>
            <key>CFBundleTypeRole</key>
            <string>Viewer</string>
            <key>NSDocumentClass</key>
            <string>MovieDocument</string>
        </dict>
    </array>
    <key>CFBundleExecutable</key>
    <string>QTKitPlayer</string>
    <key>CFBundleGetInfoString</key>
    <string>QTKitPlayer 1.0</string>
    <key>CFBundleIdentifier</key>
    <string>com.apple.QTKitPlayer</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>QTKitPlayer</string>
    <key>CFBundlePackageType</key>
    <string>APPL</string>
    <key>CFBundleShortVersionString</key>
    <string>1.0</string>
    <key>CFBundleSignature</key>
    <string>????</string>
    <key>CFBundleVersion</key>
    <string>1.0</string>
    <key>NSMainNibFile</key>
```

```
    <string>MainMenu</string>
    <key>NSPrincipalClass</key>
    <string>NSApplication</string>
</dict>
</plist>
```

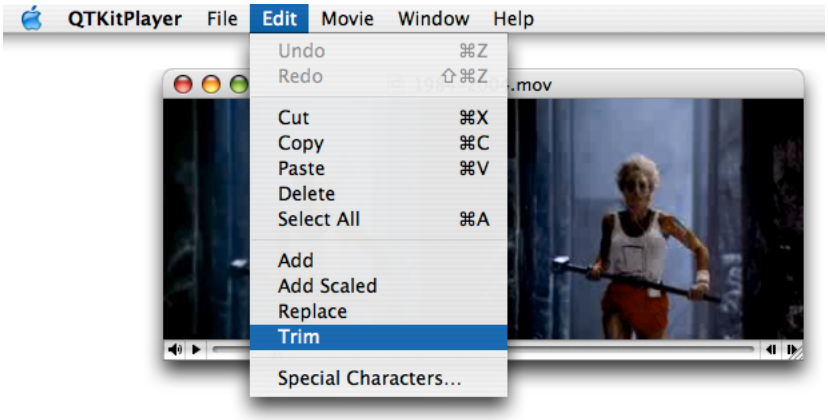
Running and Building Your QTKitPlayer

If you've followed correctly all the steps described in this chapter, you'll be ready to build and run the QTKitPlayer application. In Xcode, simply click the hammer icon with the play button on top, or press Command-R, as you would if you were compiling any other Xcode project.

The QTKitPlayer appears with the blue default movie displayed. Now you can open and import other media, export them to different formats, edit the contents of that media, and control movie playback and display, such as showing or not showing the movie controller.

You'll also be able to perform a variety of movie-editing operations discussed earlier in this chapter, such as trimming (shown in Figure 3-22).

Figure 3-22 The Apple 1984-2004 QuickTime movie with the editing trim feature enabled



Again, the Add command lets you add a new chunk to a movie in one or more new tracks––on top of what's already there––instead of inserting it like you would do using the Paste command. The Add Scaled command scales the chunk you are pasting in time, and the chunk becomes stretched or compressed so that it plays for the same duration as the current selection in the receiving movie, which can result in a slow-motion or fast-motion effects. The Trim command deletes everything in the current movie except the current selection

If you want to export your movie to another media format, such as MPEG-4, choose the Export command in the Edit menu, as shown in Figure 3-23, and the export sheet (Figure 3-24) will appear.

Figure 3-23 Exporting the Apple 1984-2004 QuickTime movie in your QTKitPlayer application

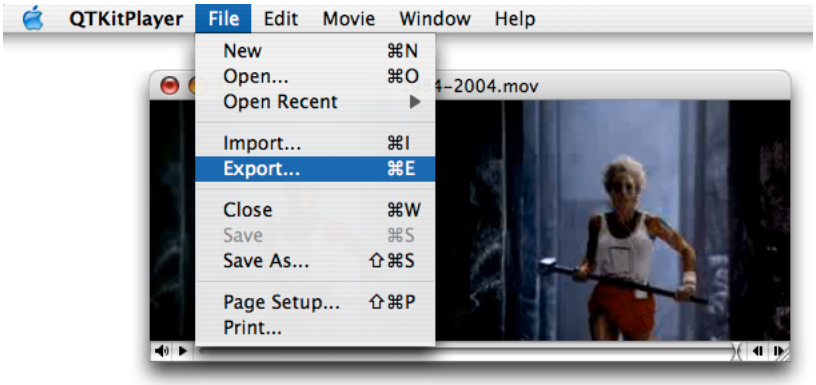
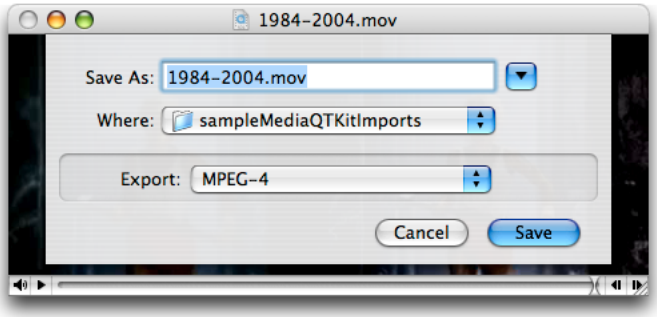


Figure 3-24 The export sheet for exporting a QuickTime movie to MPEG-4



What's Ahead?

By working through the example code in this chapter and building the extended QTKitPlayer project, you've laid the foundation for an even more powerful and robust media player application to come.

In the next chapters of this tutorial, you'll move into a more advanced learning phase. You'll add new features and capabilities to your QTKitPlayer, such as the ability to open QuickTime streaming movies from URLs. You'll also learn how to implement a Cocoa drawer with a timer that updates synchronously in real time the rate at which a QuickTime movie is displayed. In later chapters, you'll extend the QTKitPlayer to handle more robust play back of multimedia content, with up to six movies, video streams, and QuickTime virtual reality movies playing at the same time.

Each chapter is intended to demonstrate, step by step, how you can enhance your QTKitPlayer application—and in the process, learn how to tap into the awesome power of Apple's QTKit framework.

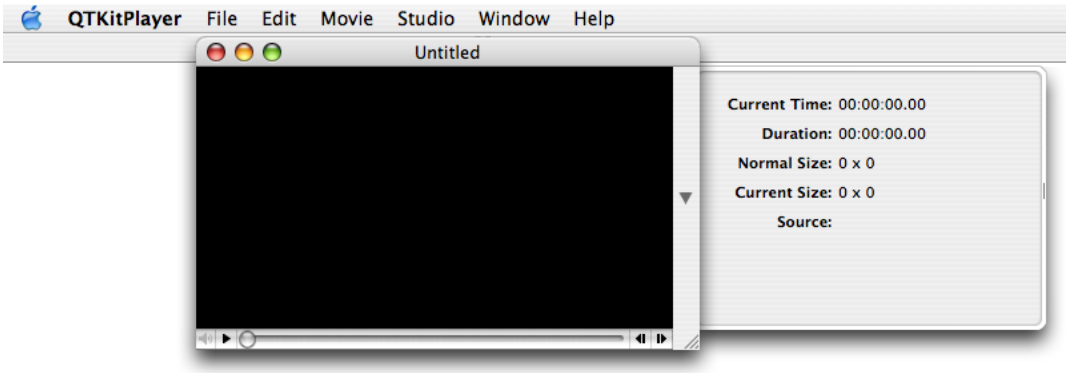
Adding New Capabilities to the QTKitPlayer Application

In this chapter, you'll extend the QTKitPlayer application beyond what you have constructed in the previous chapter. You'll add an NSDrawer object to the QTMovieView window, which toggles open and close with the click of a mouse. The drawer will also have new, enhanced functionality, including the capability of displaying a movie's current time and duration, its normal and current size, and its source.

Notably, as you work through the steps outlined in this chapter and add new chunks of code to your Xcode project, you'll implement a callback mechanism from the QuickTime C API.

When you've completed your Xcode project, you'll be able to choose File > New in your extended QTKitPlayer and an untitled QuickTime movie with a black screen will open for you. If you click the button in the middle of the frame to the right, as shown in Figure 4-1, a Cocoa drawer pops open.

Figure 4-1 An untitled QuickTime movie with an open Cocoa drawer

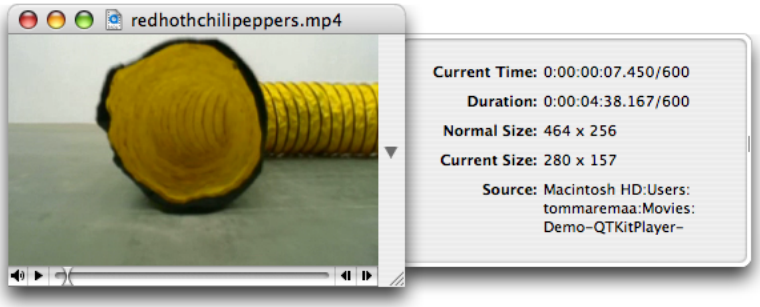


The button in the movie frame lets you toggle the drawer open and closed, with fields of useful information about the movie being displayed. The information displayed, however, is not static.

By implementing a callback mechanism from the QuickTime C API, your QTKitPlayer will be able to synchronize the current time to the playback of the movie itself.

When a QuickTime movie plays, for example, the current time field is updated synchronously with the movie, as illustrated in Figure 4-2.

Figure 4-2 An mp4 QuickTime movie playing with the current time, duration, movie size, and movie displayed in the open Cocoa drawer



Design Strategy

Building on the existing functionality of your QTKitPlayer application, you'll add new text fields to your NSDrawer object that provide information about the size of the movie playing, its source path on the user's computer, as well as the time and duration of play. These new fields may serve to enhance the user's ability, for example, to edit a QuickTime movie with a specific timeline, or to resize a movie to a particular pixel ratio. The location of the source movie may also be useful.

Because you are adding to the existing QTKitPlayer application, all the functionality of the player is still intact. All the controls for movie playback are available for starting and stopping, showing and hiding the controller, stepping forward and backward frame by frame, and so on.

This chapter depends on your having worked through the construction of the QTKitPlayer application in the previous chapter. If you haven't studied the steps described in that chapter, or simply skimmed over them, it might be a good idea to go back there first and refresh your memory before proceeding.

In this, as well as in the following chapters, you'll break down the tasks you need to accomplish into a series of discrete steps. The goal is to work as efficiently as possible by first laying the foundation for your enhanced player application in Interface Builder, and then adding the code you need to make it work.

Tasks to Accomplish

The tasks you want to accomplish in adding new functionality to your QTKitPlayer application are straightforward enough if you have already developed projects with Cocoa and Xcode. You'll do this:

1. Add a new Cocoa NSDrawer object to your project, using the tools available to you in Interface Builder and Xcode 2.0. The goal is for the user to be able to toggle the drawer open and closed, and for the drawer object to have the necessary logic to display and synchronously update the movie content as it plays.

2. Add a small chunk of code to the `MovieDocument.h` class interface to specify the movie attributes of the drawer.

3. Add a larger chunk of code to your `MovieDocument.m` implementation file. This will involve working with a callback mechanism from the QuickTime C API. The goal is to learn how you can tap into the rich library of the QuickTime C API, when you want to add specific functionality to your Cocoa project that may not be available, for whatever reason, in the Cocoa API.

Project Complexity

The complexity of the project comes with the addition of a callback mechanism from the QuickTime C API.

Basically, as the movie is playing, you need a mechanism to update the timer field in the drawer. The movie controller component calls your action filter function each time the component receives an action for its movie controller. In your action filter, you will use the idle action (`mcActionIdle`) to update your time display. The idle action is sent continuously while the movie is being serviced.

You could do a similar thing with an `NSTimer`, but the action filter technique is a bit easier to implement.

The function prototype for the QuickTime C callback mechanism is defined as follows:

```
Boolean MyMCActionFilterWithRefConProc
        (   MovieController      mc,
            short               action,
            void                *params,
            long                refCon );
```

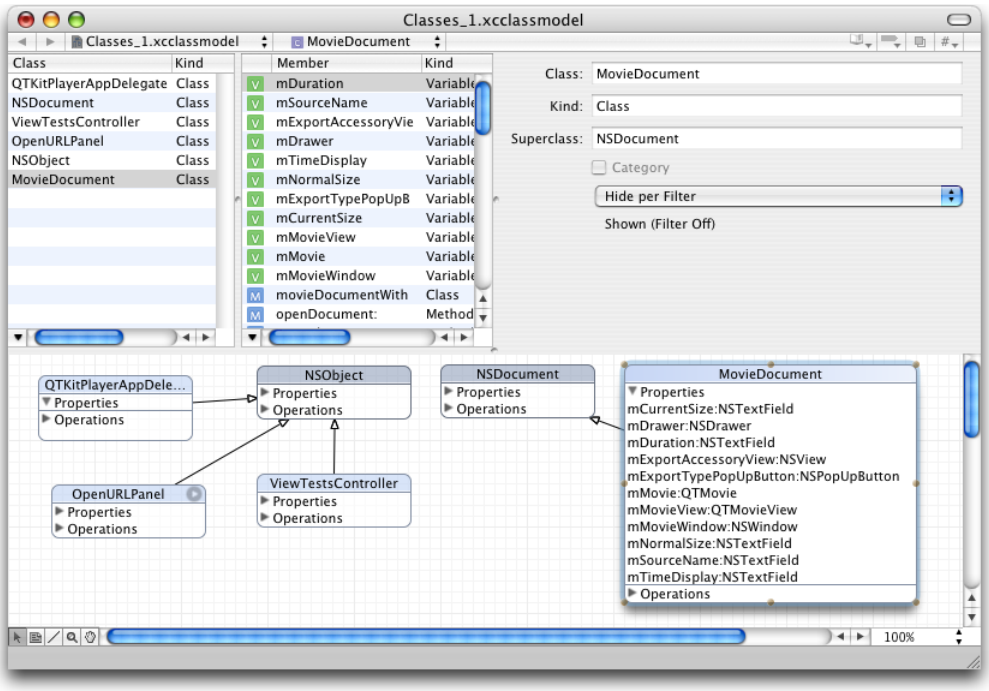
The `MyMCActionFilterWithRefConProc` function responds to the actions of a movie controller with a reference constant. The parameters specify the movie controller (*`mc`*) for the operation, the movie controller action (*`action`*), a pointer to a structure that passes information to the callback, and a reference constant (*`refCon`*) that your code supplies to your callback. Your QTKitPlayer application can invoke these actions by calling `MCDoAction`. An action filter function will receive any of the controller actions sent to it.

You'll add this callback to the code in your `MovieDocument.m` implementation file. The way to accomplish this is explained in detail in the section "Adding Code To Make the Drawer Work" (page 69).

Class Model

When you've completed all these tasks, you'll see the class model of your QTKitPlayer code in Xcode 2.0, as shown in Figure 4-3. The class model includes a list of the MovieDocument instance variables you'll add to your QTKitPlayer project and a visualization of their connections and inheritance paths.

Figure 4-3 A class model of the MovieDocument class with a list of properties



Note that this class model diagram is only available using Xcode 2.0. As illustrated in Figure 4-3, it also lists the classes you'll work with in subsequent chapters of this programming tutorial. Just ignore these for the moment, as you focus on the properties you'll add to the MovieDocument class.

Adding a Cocoa Drawer to the QTKitPlayer

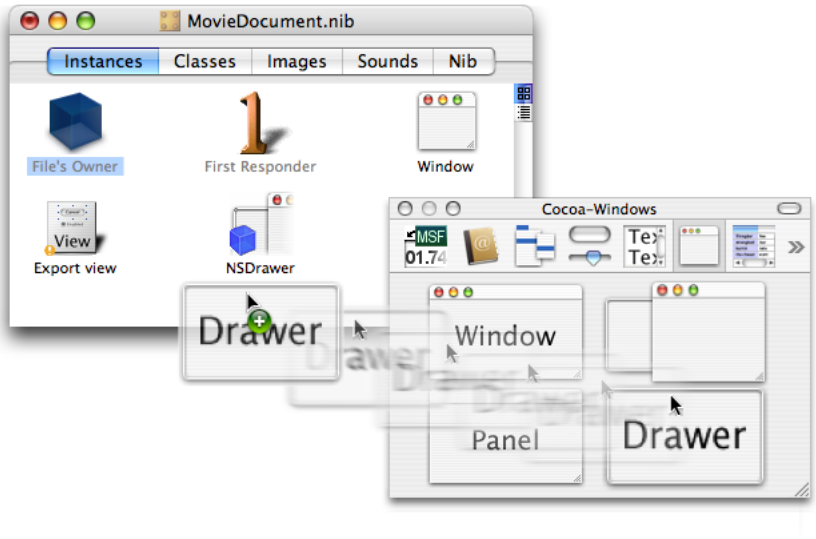
In this sequence of steps, you'll add a Cocoa drawer to the QTKitPlayer. Adding drawers to a window is relatively easy to accomplish using Interface Builder.

To extend your QTKitPlayer, follow these steps:

1. Launch Interface Builder 2.5 and open your MovieDocument.nib.

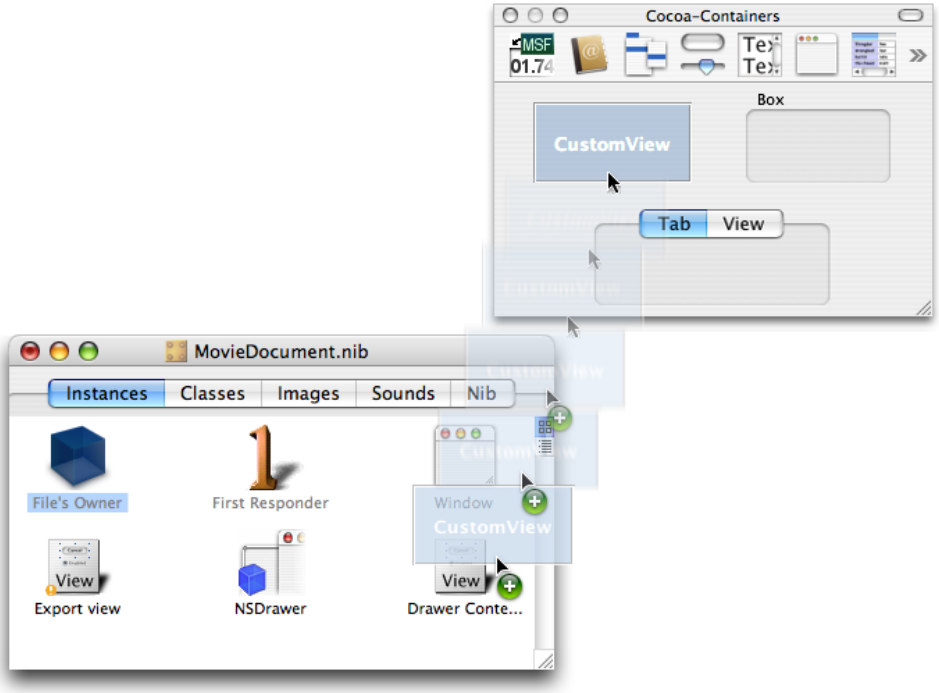
2. Drag the drawer object from your Cocoa-Windows palette into your nib, shown in Figure 4-4. The drawer object becomes an NSDrawer in your nib.

Figure 4-4 The Cocoa-Windows and the NSDrawer object added in MovieDocument.nib



3. To add the drawer content view, drag a CustomView object from the Cocoa-Containers palette into the nib. Double-click the icon and rename it as "Drawer ContentView". That gives you the content view, as shown in Figure 4-5.

Figure 4-5 The CustomView object in the Cocoa-Containers palette and in the MovieDocument.nib as Drawer ContentView

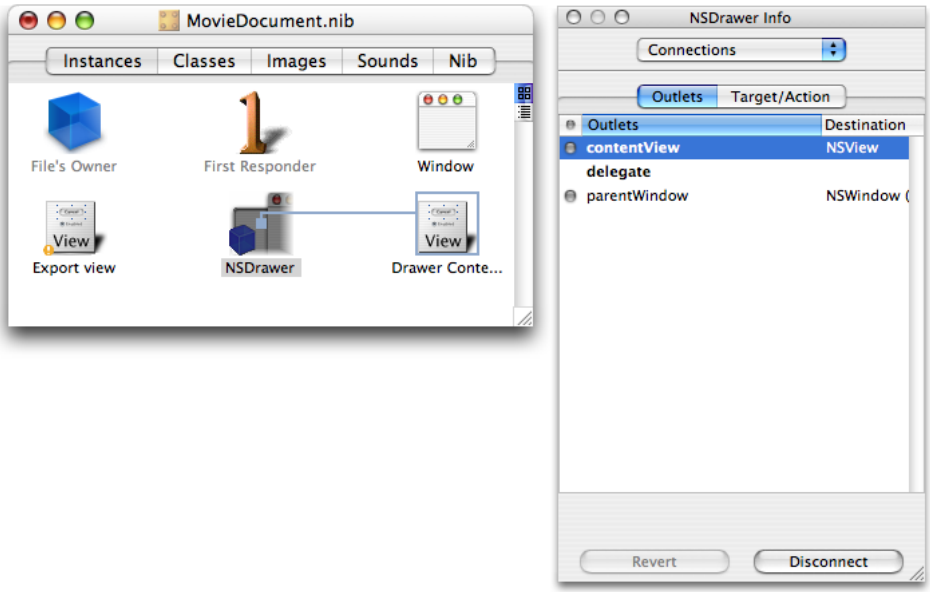


4. Add the `mDrawer` outlet to File's Owner. Double-click the File's Owner icon. This brings up the Attributes pane for the MovieDocument Info window. Click Add to add an `mDrawer` outlet.

5. Wire up the connection from the File's Owner to the `mDrawer` outlet. Click File's Owner, press the Control key, and drag wire to the NSDrawer icon in the MovieDocument.nib. Click connect.

6. Now you want to select the NSDrawer object in your nib and press Command-2 to open the NSDrawer Info pane. Select the NSDrawer icon and press the Control key. Now Control-drag to the Drawer Content View icon and press the Connect button in the NSDrawerInfo window, as shown in Figure 4-6. You want to wire up your connections from the NSDrawer object to the Drawer Content View object.

Figure 4-6 The NSDrawer wired up and connected to the contentView outlet



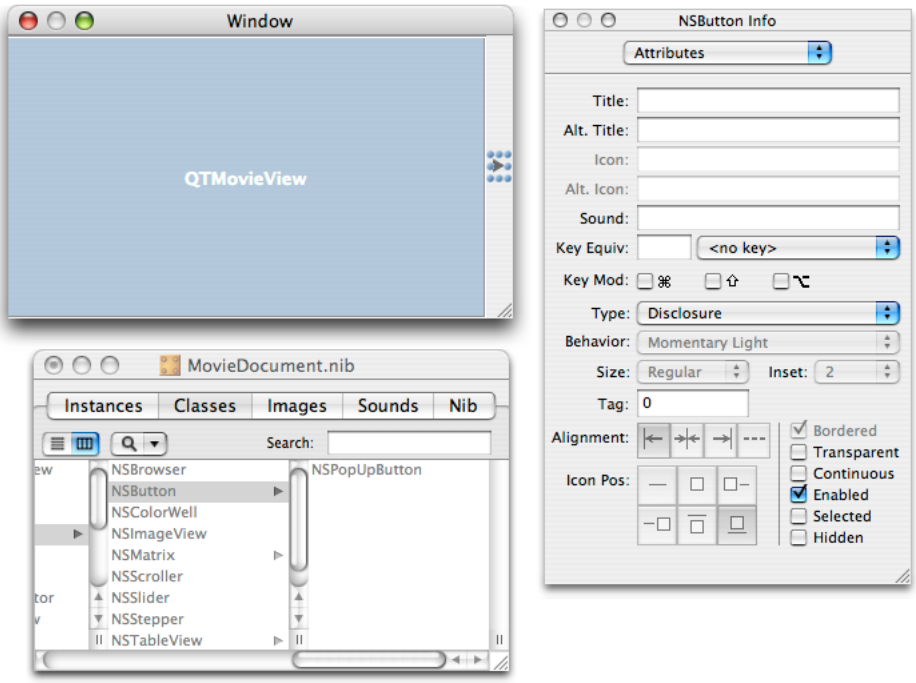
7. Similarly, select the NSDrawer icon and press the Control key. Now control-drag to the window icon to wire up your connections from the NSDrawer object to its parent window.

8. In the MovieDocument.nib, double-click the window icon to select the QTMovieView window object. You want to leave a portion of the window available for the button that will toggle your NSDrawer object open and closed.

9. Select the Cocoa-Controls palette and drag an NSButton object from the collections of control object to the right portion of your window.

10. Press Command-1 to open the Attributes pane, as shown in Figure 4-7.

Figure 4-7 The NSButton info pane with attributes and the type specified as disclosure

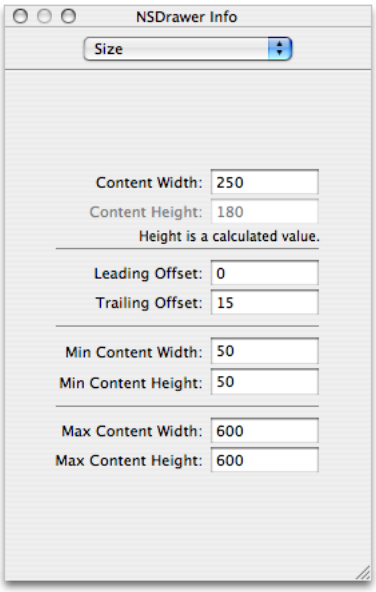


11. Specify the Type as Disclosure in the attributes pane of NSButton Info (Figure 4-7). This enables the button to toggle. The toggle option is available in Cocoa by enabling disclosure.

12. Set the springs for the disclosure button.

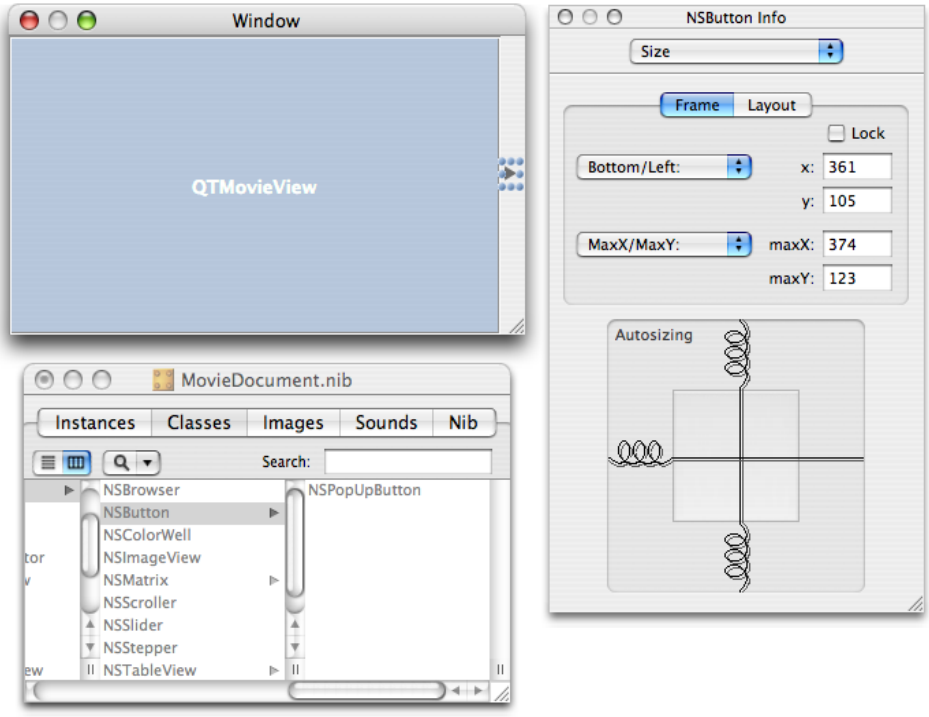
13. Click the NSDrawer icon, then set the Size attributes in the NSDrawer Info pane, as shown in Figure 4-8.

Figure 4-8 Size attributes added to the Drawer Content View



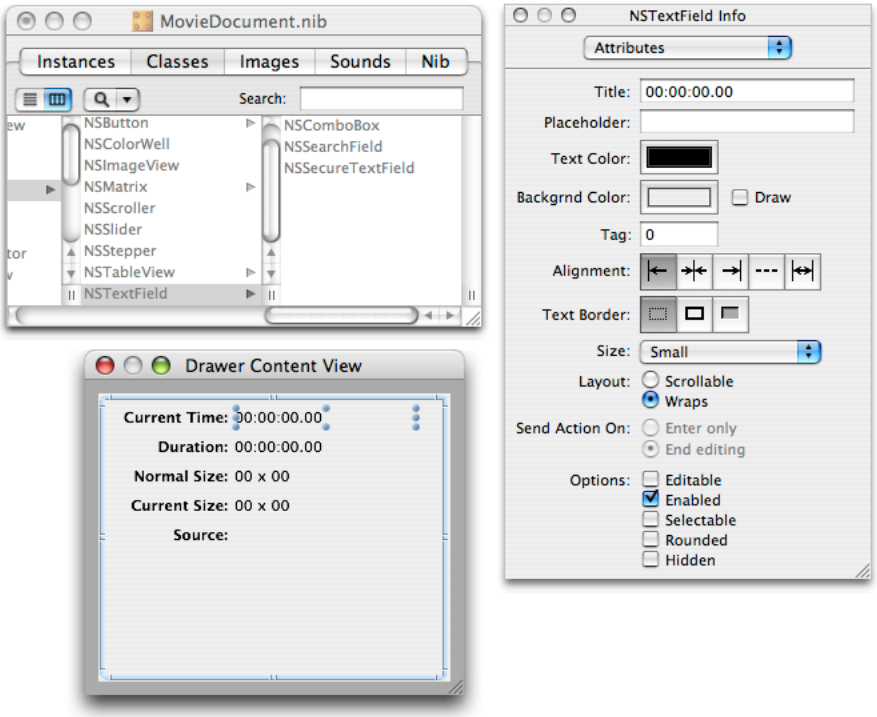
14. Now you want to set the springs in the Size attribute pane so that when the user resizes the drawer, the toggle button is not crunched. Set the springs for autosizing, as shown in Figure 4-9.

Figure 4-9 The Size settings for autosizing of springs in the Drawer Contents object



15. Add NSTextField objects corresponding to "Current Time," "Duration," "Normal Size," "Current Size," and "Source" (all static text), as shown in Figure 4-10. In the attributes pane of each text field, specify the title, color, alignment, text border, and other information.

Figure 4-10 Text fields and their attributes added to the Drawer Content View

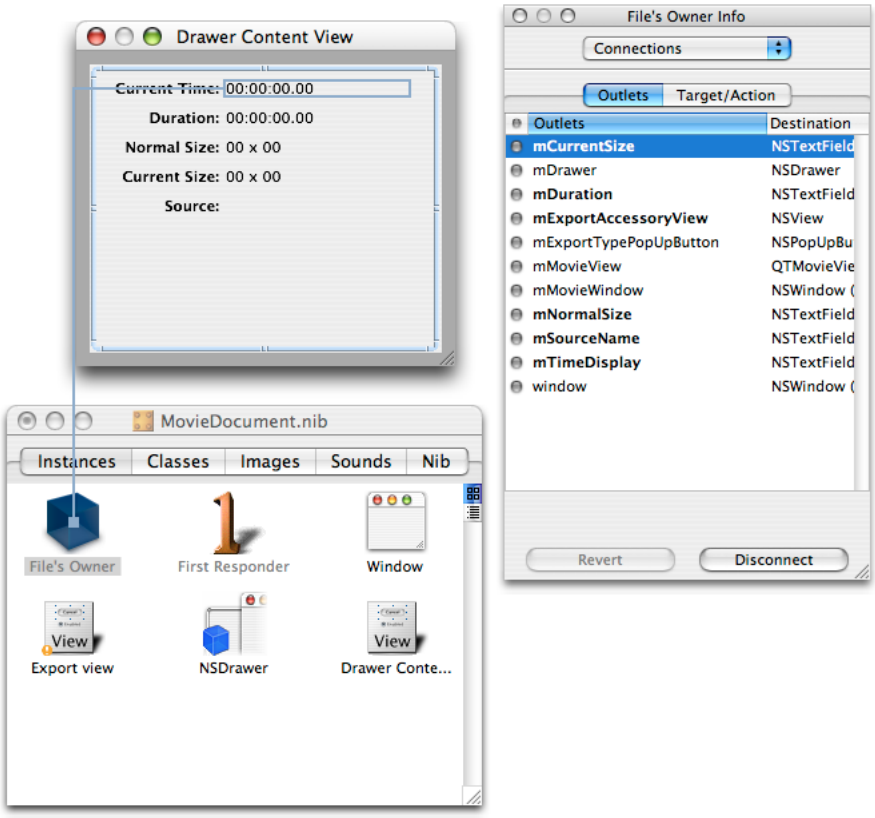


16. You want to be sure that you specify in the NSTextfield Info panes for Current Time and Duration by entering the Title attribute as 00:00:00:00, as shown in Figure 4-10.

17. Now add the outlets to your document subclass. Double-click the File's Owner icon. This brings up the Info window attributes pane for the MovieDocument class. Use the Add button and add the following outlets: `mCurrentSize`, `mDuration`, `mNormalSize`, `mSourceSize`, and `mTimeDisplay`.

18. Now you are ready to hook up the File's Owner to the various textfields that you have specified in the Drawer Content View. Select the File's Owner icon and press the Control key. Connect the wire to each textfield in the Drawer Content View, as shown in Figure 4-11 and click Connect for each outlet.

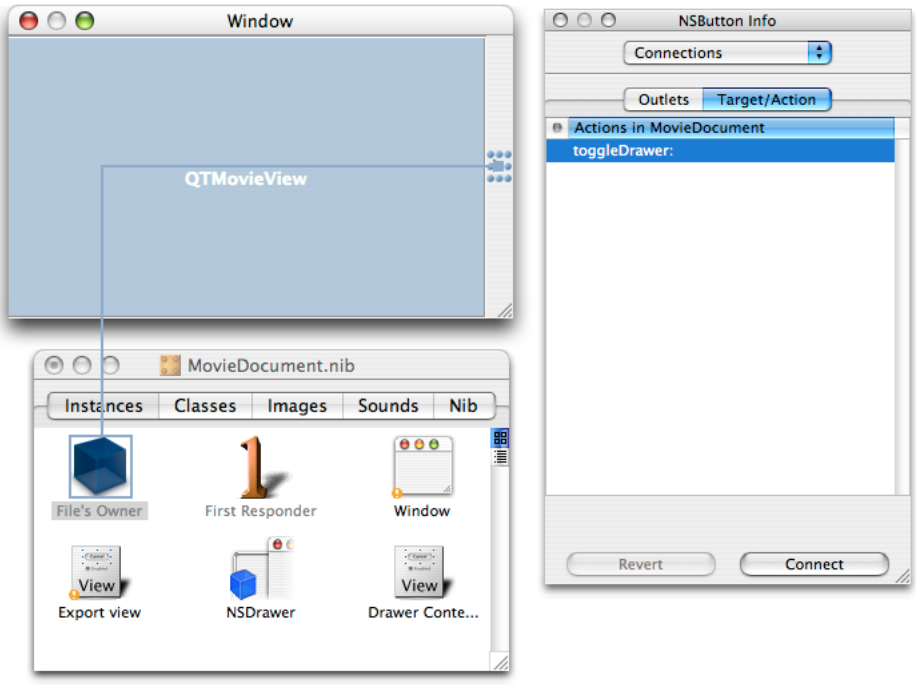
Figure 4-11 Hooking up the textfields and adding outlets



19. Add toggle drawer action to the MovieDocument class. Double-click File's Owner, and in the Attributes pane of the Info window for the MovieDocument, click the "O Actions" pane. Now click the Add button, and add a `toggleDrawer` action.

20. Wire up the disclosure button to the `toggleDrawer` action. Click the disclosure button in the MovieDocument window. Press the Control key, then drag the wire to the File Owner's icon. In the Connections pane of the Info window for NSButton, click the Target/Action pane. Select the `toggleDrawer` action and click the Connect button, as shown in Figure 4-12.

Figure 4-12 Wiring up the toggle drawer target and action in the MovieDocument.nib



This completes the steps you need to follow in order to construct the Cocoa drawer and hook up the outlets and actions in Interface Builder.

Adding Code To Your QTKitPlayer Project

In this section, you'll add less than one hundred lines of code to your QTKitPlayer project by modifying your `MovieDocument.h` and `MovieDocument.m` class files.

Adding Code To The Movie Document Class

In this next sequence of steps, you'll be adding a small amount of code to your `MovieDocument.h` class interface file. You want to declare the instance variables that you have specified as your outlets and target actions in Interface Builder. You also want to define in code the methods that will enable you to set these variables.

To begin, open the `MovieDocument.h` declaration file in your Xcode project. Follow these steps to add to your existing code in the file:

1. Insert the following line of code in the `@interface` directive block after your export outlet declarations. This enables you to specify the movie attributes for your NSDrawer object:

```
IBOutlet NSDrawer  *mDrawer;
```

2. Add the following code (also in the `@interface` block) to specify the drawer elements for your movie attributes:

```
 IBOutlet NSTextField *mCurrentSize;
    IBOutlet NSTextField *mDuration;
    IBOutlet NSTextField *mNormalSize;
    IBOutlet NSTextField *mSourceName;
    IBOutlet NSTextField *mTimeDisplay;
```

3. Define a getter for your drawer with the following line of code:

```
- (id)drawer;
```

4. Add to your IBActions to toggle the drawer open and closed, with the following line of code:

```
- (IBAction)toggleDrawer:(id)sender;
```

5. Insert the following methods before the `@end` directive in the file:

```
- (void)setDurationDisplay;
- (void)setNormalSizeDisplay;
- (void)setCurrentSizeDisplay;
- (void)setSource:(NSString *)name;
- (void)setTimeDisplay;
```

6. Add these two methods to install and remove the movie callback before the `@end` directive in the file:

```
-(void)removeMovieIdleCallback;
-(void)installMovieIdleCallback;
```

Adding Code To Make the Drawer Work

In this next sequence of steps, you'll be adding a larger chunk of code to your `MovieDocument.m` implementation file.

To begin, open the `MovieDocument.m` file in your Xcode project. You'll add the following blocks of code to your file. Just follow these steps:

1. After your import and `#define` statements, add this line of C code to set your timer display. You pass your movie document in the `refcon` parameter and that gives you a way to call the `setTimeDisplay:` method. This is the function prototype you add:

```
pascal Boolean MyActionFilter (MovieController mc, short action, void* params,
 long refCon);
```

   As the movie is being serviced by QuickTime, the movie controller component calls your action filter procedure each time the component receives an action for its movie controller. You use the idle action in your filter procedure to update your timer display.

2. Next, you want to add these initializations for your movie drawer items. Insert these lines:

```
-(void)awakeFromNib
{
    // initialize movie drawer items
    [self setDurationDisplay];
    [self setNormalSizeDisplay];
    [self setCurrentSizeDisplay];
    [self setSource:[self fileName]];
}
```

3. You want to return the `mDrawer` instance variable. Insert these lines before you add your setters, in a block identified as Getters:

```
- (id)drawer
{
    return mDrawer;
}
```

4. Now you want to add the following block of code, which enables you to read the movie. This is also where you install the movie action callback. Add these lines:

```
- (BOOL)readFromFile:(NSString *)fileName ofType:(NSString *)type
{
    BOOLsuccess = NO;

    // read the movie
    if ([QTMovie canInitWithFile:fileName])
    {
        if ([type isEqualTo:@"MovieDocumentData"])
        {
            NSData*data = [NSData dataWithContentsOfFile:fileName];
            [self setMovie:[QTMovie movieWithData:data error:nil]];
        }
        else
        {
            [self setMovie:((QTMovie *)[QTMovie movieWithFile:fileName
error:nil])];
        }

        success = (mMovie != nil);

        if (success)
        {
            [self installMovieIdleCallback];
        }

    }

    return success;
}
```

5. Next, you need to install a movie controller action filter. If the movie is successfully created, the callback is installed. To get the controller from the movie and set the movie control filter, add this block of code:

```
-(void)installMovieIdleCallback
{
    ComponentResult cr = noErr;

    MovieController mc = [mMovie quickTimeMovieController];
```

```
    if (!mc) goto bail;

    MCActionFilterWithRefConUPP upp =
NewMCActionFilterWithRefConUPP(MyActionFilter);
    if (!upp) goto bail;

    cr = MCSetActionFilterWithRefCon(mc, upp, (long)self);
    DisposeMCActionFilterWithRefConUPP(upp);

bail:
    return;
}
```

6. When the movie is closed, you remove the existing callback by passing `NIL` for the callback parameter. Add these lines of code:

```
-(void)removeMovieIdleCallback
{
    [MCSetActionFilterWithRefCon([mMovie quickTimeMovieController],
                                 nil,
                                 (long)self);
}
```

7. Insert the following code to toggle the drawer state:

```
- (IBAction)toggleDrawer:(id)sender
{
    [mDrawer toggle:sender];
```

8. In the next sequence of steps (8-12), you add these chunks of code to the fields in the drawer. To set the current time text field, insert this chunk of code:

```
- (void)setTimeDisplay
{
    if (mMovie)
    {
        QTTime currentPlayTime = [[mMovie
attributeForKey:QTMovieCurrentTimeAttribute] QTTimeValue];
        [mTimeDisplay setStringValue:QTStringFromTime(currentPlayTime)];
    }
}
```

9. To set the duration text field, insert this chunk of code:

```
- (void)setDurationDisplay
{
    if (mMovie)
    {
        [mDuration setStringValue:QTStringFromTime([mMovie duration])];
    }
}
```

10. To set the normal size text field, add this:

```
- (void)setNormalSizeDisplay
{
    NSMutableString *sizeString = [NSMutableString string];
    NSSize movSize = NSMakeSize(0,0);
```

```
    movSize = [[mMovie attributeForKey:QTMovieNaturalSizeAttribute] sizeValue];

    [sizeString appendFormat:@"%.0f", movSize.width];
    [sizeString appendString:@" x "];
    [sizeString appendFormat:@"%.0f", movSize.height];

    [mNormalSize setStringValue:sizeString];
}
```

11. To set the current size text field, add:

```
- (void)setCurrentSizeDisplay
{
    NSSize movCurrentSize = NSMakeSize(0,0);
    movCurrentSize = [[mMovie attributeForKey:QTMovieCurrentSizeAttribute]
sizeValue];
    NSMutableString *sizeString = [NSMutableString string];

    if (mMovie && [mMovieView isControllerVisible])
        movCurrentSize.height -= [mMovieView controllerBarHeight];

    [sizeString appendFormat:@"%.0f", movCurrentSize.width];
    [sizeString appendString:@" x "];
    [sizeString appendFormat:@"%.0f", movCurrentSize.height];

    [mCurrentSize setStringValue:sizeString];
}
```

12. To set the source text field, add:

```
- (void)setSource:(NSString *)name
{
    NSArray *pathComponents = [[NSFileManager defaultManager]
componentsToDisplayForPath:name];
    NSEnumerator *pathEnumerator = [pathComponents objectEnumerator];
    NSString *component = [pathEnumerator nextObject];
    NSMutableString *displayablePath = [NSMutableString string];

    while (component != nil) {
        if ([component length] > 0) {
            [displayablePath appendString:component];

            component = [pathEnumerator nextObject];
            if (component != nil)
                [displayablePath appendString:@":"];
        } else {
            component = [pathEnumerator nextObject];
        }
    }

    [mSourceName setStringValue:displayablePath];
}
```

13. After the `@end directive`, you add this chunk of code to intercept some actions for the movie controller and to update the current time display in the drawer. Note that the `refCon` parameter is a document `id`:

```
pascal Boolean MyActionFilter (MovieController mc, short action, void* params,
 long refCon)
{
    MovieDocument *doc = (MovieDocument *)refCon;

    switch (action)
    {
        // handle idle events
        case mcActionIdle:
            // update the current time display in the info drawer
            if ([[doc drawer] state] != NSDrawerClosedState)
                [doc setTimeDisplay];
            break;
    }

    return false;
}
```

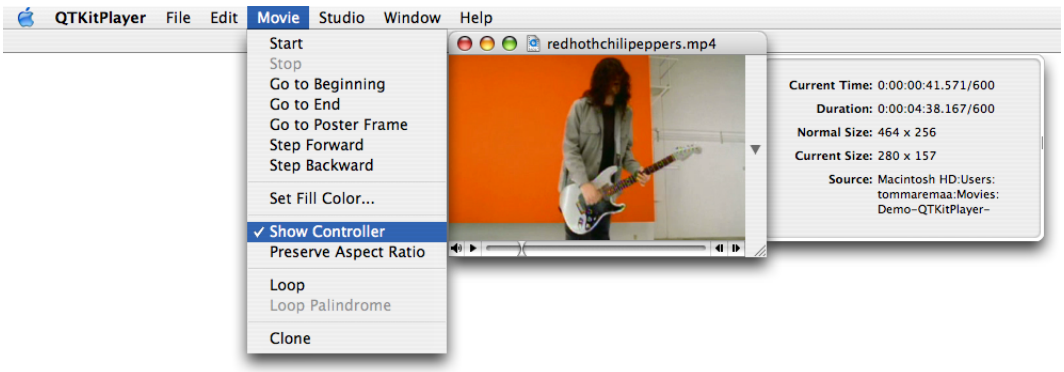
This wraps up the additions to the code you need to make in your MovieDocument implementation file. You're now ready to build and compile your Xcode project.

The Completed Project

If you've followed the steps outlined in this chapter, you'll have successfully extended the QTKitPlayer application with a Cocoa drawer you can toggle open and closed. The drawer will display and update the time of any QuickTime movie you want to play.

You can control movie playback as you have before, with QTKitPlayer functionality still in place for movie control, as shown in Figure 4-13.

Figure 4-13 Open drawer with movie Show Controller selected



In a short series of steps, you've significantly extended the capabilities of your QTKitKPlayer application. Now you're ready to enhance the media player by adding the capability to stream audio and video. That's explained in the next chapter.

Extending the QTKitPlayer To Stream Audio and Video

If you've been working diligently through the code examples in the last two chapters, you've no doubt advanced your knowledge of how to extend your QTKitPlayer application with the tools available in Xcode and Interface Builder. The goal has been to enhance the functionality of the media player by tapping into the power of the QTKit framework and in the process learn by doing.

In this chapter, you'll jump into another level of programming skill by adding the capability of streaming audio and video in your application. Streaming, of course, is not the same thing as downloading a QuickTime movie file.

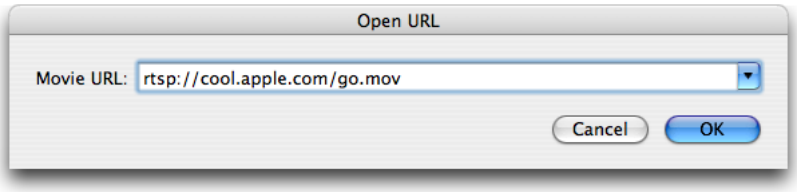
Note: Streams, which can originate from a live source or stored video on a server, are data packets that are assembled into sound and video, displayed briefly, then discarded. No movie file is created on the user's computer. The way streaming works is that a streaming server responds to requests for streaming movies, and then transmits those streams to users in response to requests. The requests are handled using RTSP (Real-Time Streaming Protocol); streams are sent over the Internet using RTP (Real-Time Transport Protocol).

The URL for a streaming movie starts with the RTSP protocol identifier and looks like this

```
rtsp://YourStreamServer.com/YourPath/YourMovie.mov
```

After you complete the steps in this chapter, your QTKitPlayer application will be able to play audio and video streams in real time over the Internet. You'll be able to open a streaming movie by choosing Open URL from the File menu and entering the URL string, as shown in Figure 5-1.

Figure 5-1 An Open URL dialog in the completed QTKitPlayer application



Clicking OK launches an Untitled QTKitPlayer movie window with a big blue Q at the center, as shown in Figure 5-2.

Figure 5-2 The movie window launched by the QTKitPlayer to stream audio and video with the big blue Q at its center

When you click the play button in the control bar at the lower left, your application will send a request to a server to fetch the audio or video stream with the RTSP protocol identifier. A few seconds later, the stream will appear in the player window. You can then resize the window accordingly, as you wish.

You'll follow the same procedure as defined in the previous chapter: identify the tasks to accomplish, construct and wire together the necessary objects for your user interface, and then add the code to make it work.

Tasks to Accomplish

The tasks you need to accomplish to extend the capability of your streaming QTKitPlayer are more complex than those described in the previous chapter. You'll build on the existing QTKitPlayer application, using the tools available to you in Interface Builder 2.5 and Xcode 2.0. But you'll also be adding more code to the project to deal with the handling of URL streams. You'll do this:

1. Construct an OpenURL panel in Interface Builder with a ComboBox object for entering a movie URL.
2. Add actions and outlets to the OpenURL panel you've constructed.
3. Add a new menu item to the File menu to open the URL.
4. Subclass NSObject with a QTKitAppDelegate class in your MainMenu.nib and wire it up to handle the opening of a URL in your OpenURL panel.
5. Add to your Xcode project a new `OpenURLPanel.h` declaration file in which you define the class methods, getters and setters, delegates, notifications, and actions for your URL panel.
6. Add an `OpenURLPanel.m` implementation file in which you get the URL from a string, validate and save the URL, and inform the delegate before closing down the URL panel.
7. Add an `QTKitPlayerAppDelegate.h` class interface file to your project with IBActions for opening the URL panel.
8. Add an `QTKitPlayerAppDelegate.m` implementation file to your project in order to create the movie document from the URL.

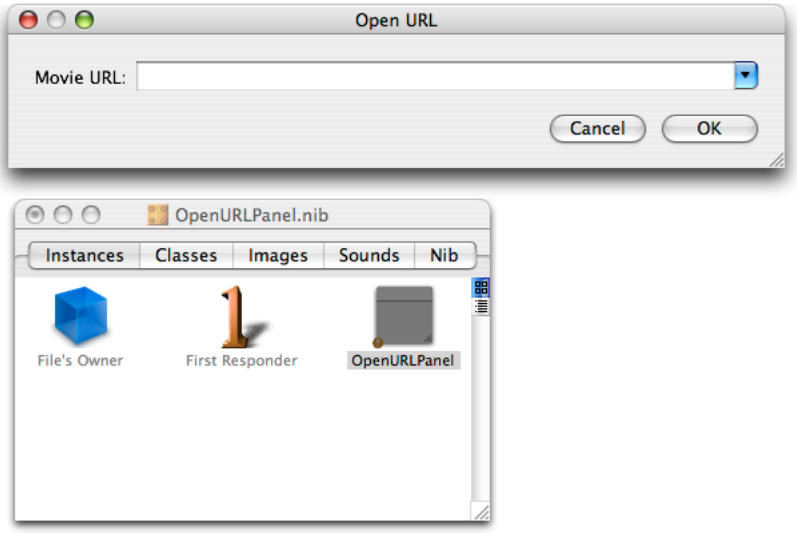
Each task is outlined, step by step, in the next sections of this chapter. You'll start as you have before with Interface Builder and then move on to add the code you need in four separate Xcode files.

Constructing the Open URL Panel

By now you should be familiar with how to work with Interface Builder and its various palettes, icons, and objects. For purposes of simplicity and to move things forward a bit faster, this means combining a few basic steps in constructing your Open URL Panel. To start:

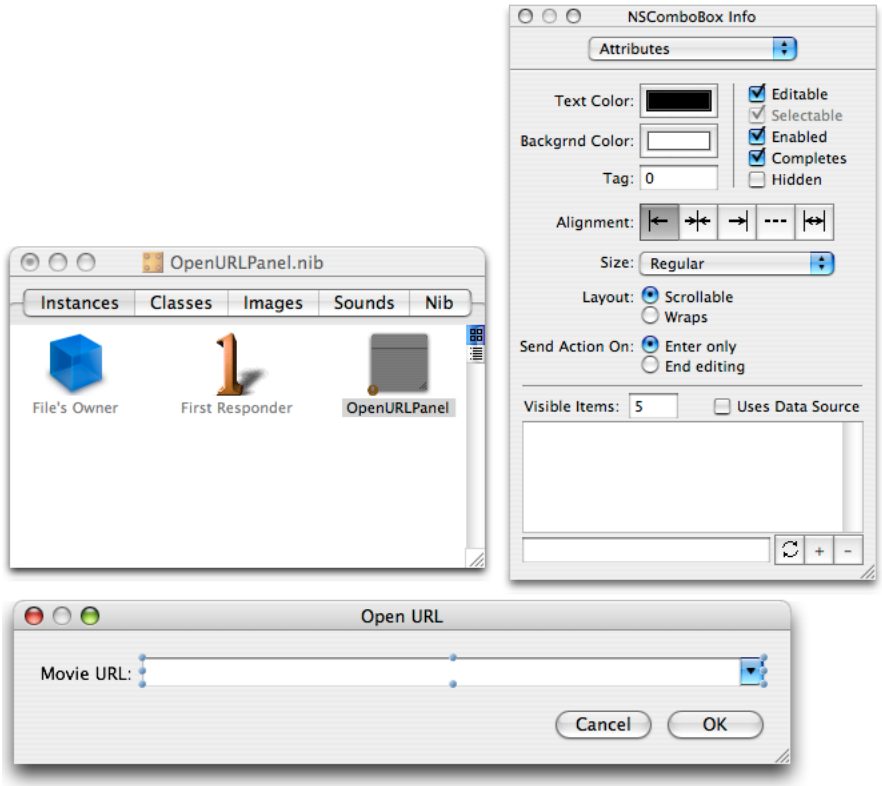
1. Create a new nib and call it OpenURLPanel.nib.
2. Create an Open URL panel object with Cancel and OK buttons, which are positioned according to Human Interface Guidelines, as shown in Figure 5-3.
3. Create a ComboBox object from the Cocoa-Data palette, where the user can enter a movie URL, as also shown in Figure 5-3.

Figure 5-3 The Open URL panel nib and dialog for entry of a movie URL



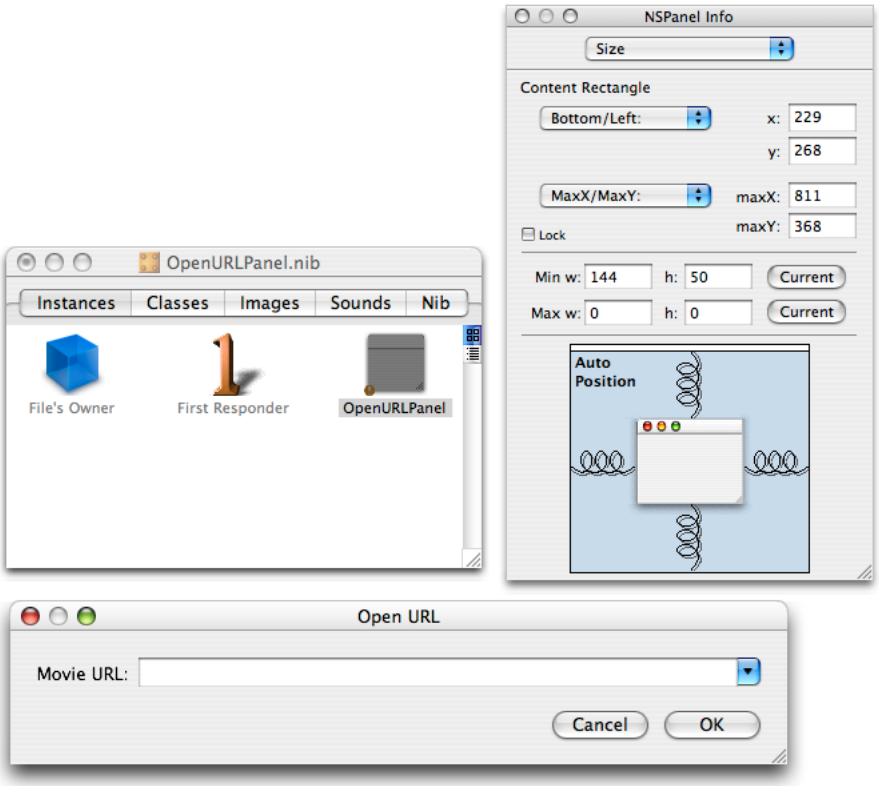
4. In the ComboBox, specify the attributes shown in Figure 5-4.

Figure 5-4 The ComboBox attributes



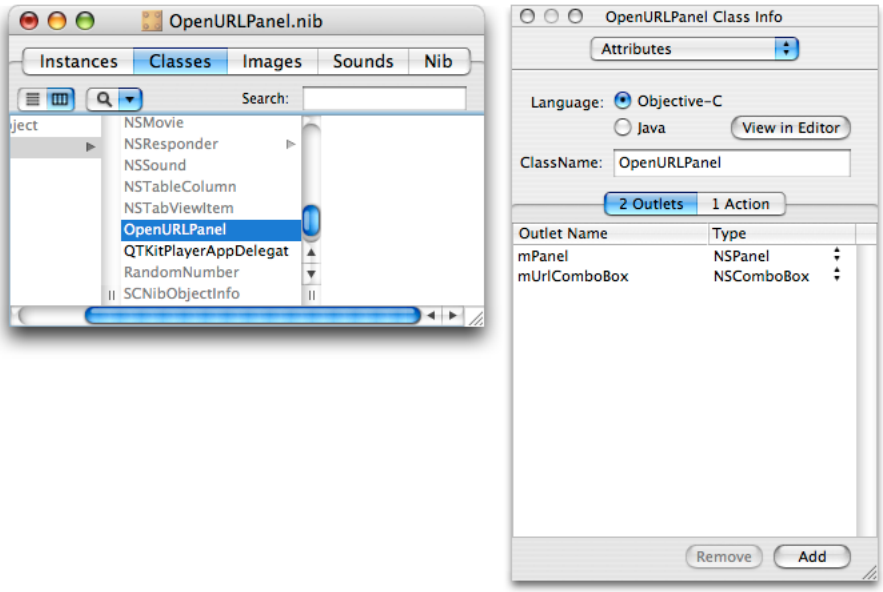
5. In the Size pane of the OpenURLPanel object, set the springs and the size of the panel, as shown in Figure 5-5.

Figure 5-5 The size settings and spring positions in the OpenURLPanel object



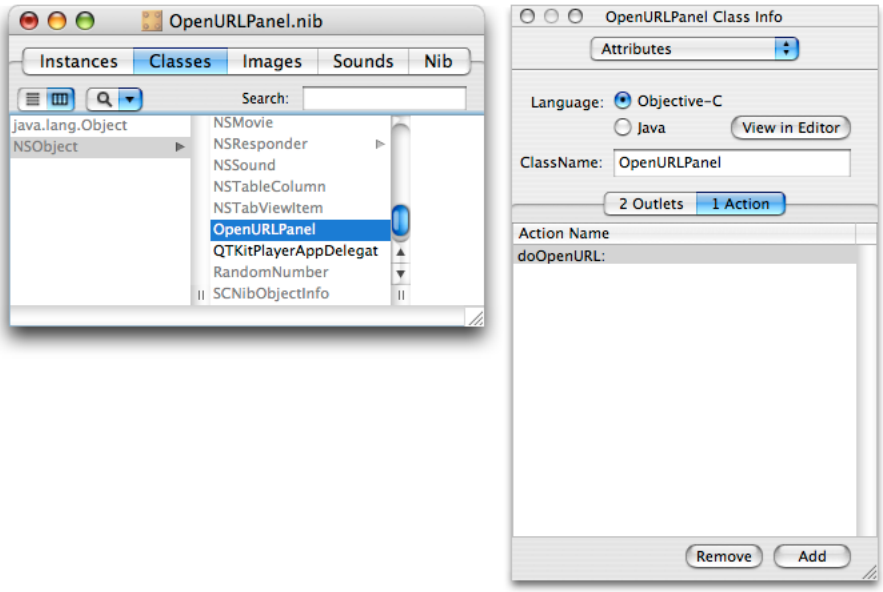
6. Now in the OpenURLPanel.nib, select OpenURLPanel from the list of classes. You can type "OpenURLPanel" in the search box and press return, or click NSObject to locate the OpenURLPanel class. Add `mPanel` and `mUrlComboBox` as your outlets, as shown in Figure 5-6.

Figure 5-6 The outlets for the OpenURLPanel class



7. Now add one action to the the OpenURLPanel class in the Attributes pane, as shown in Figure 5-7.

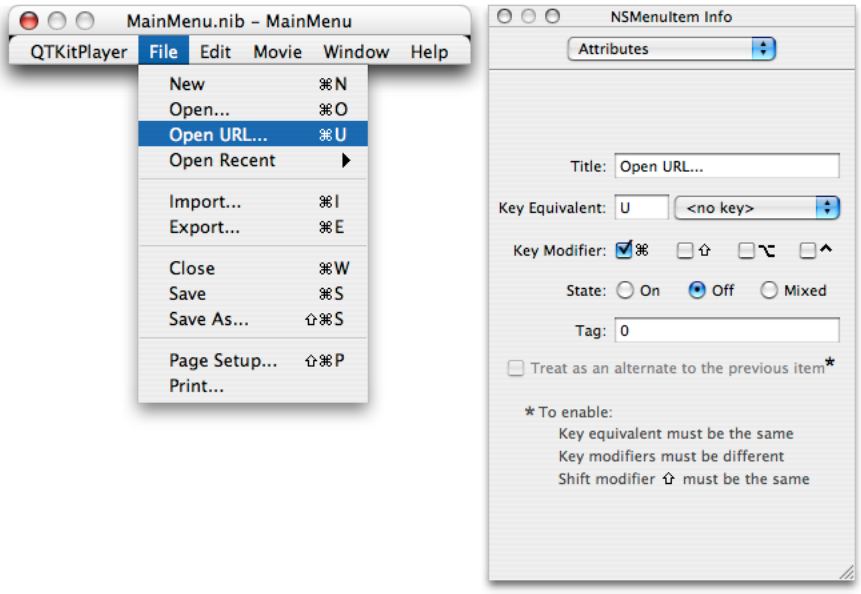
Figure 5-7 Action added to the OpenURLPanel



8. Save the nib and open the MainMenu.nib - MainMenu.

9. Add a menu item in the File menu as Open URL. . . with a key equivalent of Command-U, and specify its attributes as shown in Figure 5-8.

Figure 5-8 The main menu nib with the menu attributes specified

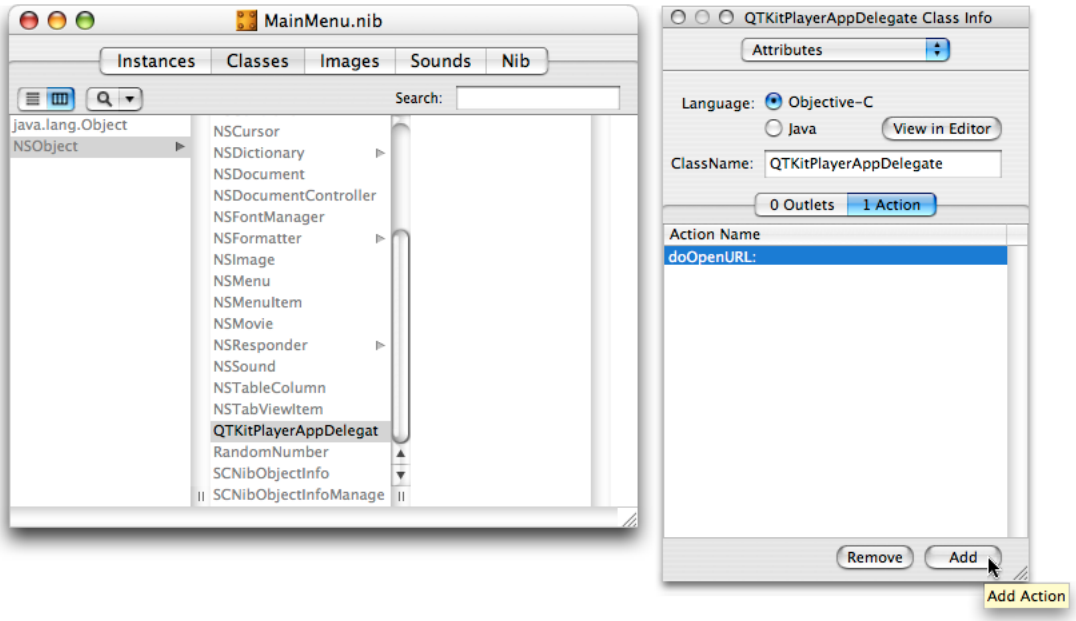


10. Now subclass NSObject and create a new class called `QTKitPlayerAppDelegate`.

11. Select `QTKitPlayerAppDelegate` from the list of classes in your MainMenu.nib and open the Attributes pane.

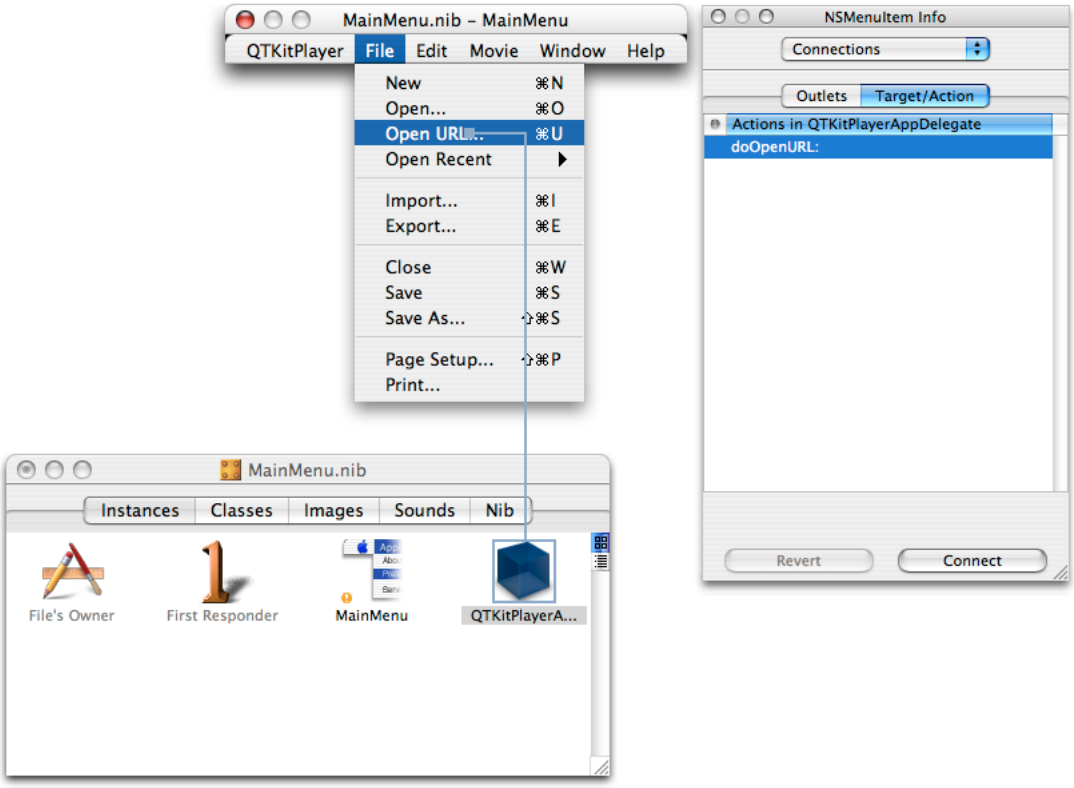
12. Add the `doOpenURL` action to your `QTKitPlayerAppDelegate`, as shown in Figure 5-9.

Figure 5-9 Actions added to the QTKitPlayAppDelegate class in the Attributes pane



13. Now add the wiring to connect the OpenURL menu item and its action `doOpenURL` to the `QTKitPlayAppDelegate` class, as shown in Figure 5-10.

Figure 5-10 Wiring to nib



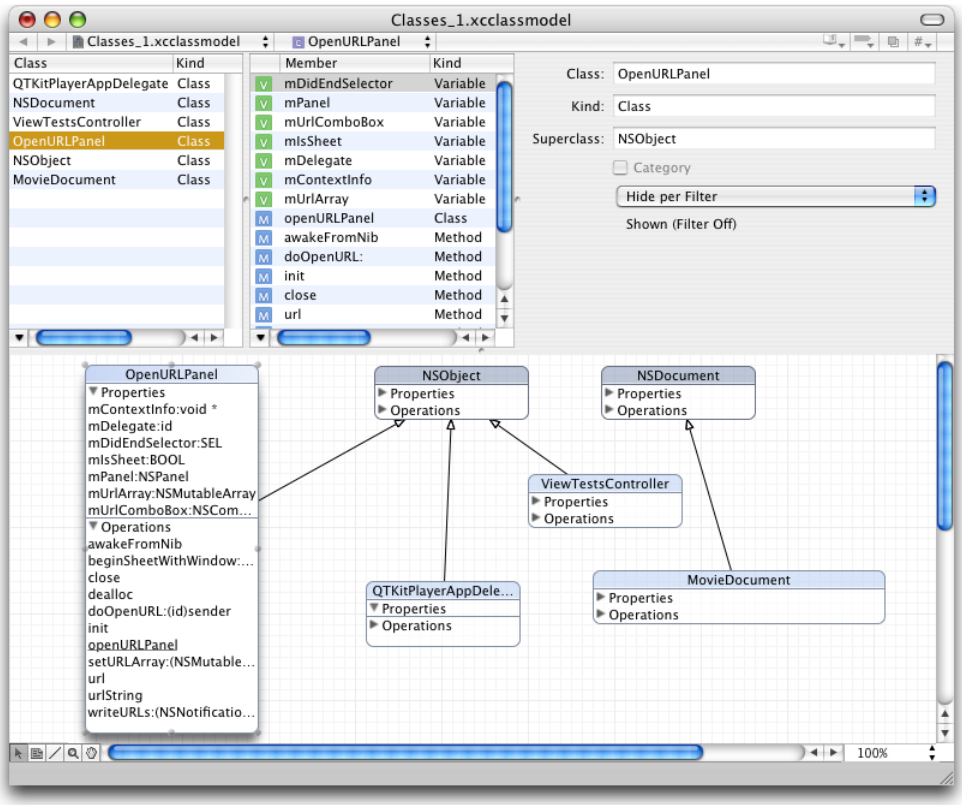
14. Save the files in Interface Builder and Quit.

If you've completed the steps outlined this section, you'll be ready now to add the necessary code to your project.

Adding Code To Stream Audio and Video

In this section, you'll add four new files to your Xcode project, including a `OpenURLPanel.h` declaration file, an `OpenURLPanel.m` implementation file, a `QTKitPlayerAppDelegate.h` class interface file, and a `QTKitPlayerAppDelegate.m` implementation file. Figure 5-11 shows the class model for the `OpenURLPanel` class with its properties and their connections listed.

Figure 5-11 The class model for the OpenURLPanel class



Adding Code For The OpenURLPanel Class Interface

In this next sequence of steps, you'll add the code you need for your `OpenURLPanel.h` class interface file.

To begin, in your QTKitPlayer project, choose File > New File. In the Assistant window for your new file in Xcode 2.0, select Cocoa > Objective-C class and in the window that opens enter the title `OpenURLPanel.h`. Now follow these steps:

1. Insert the following import code at the beginning of your file:

```
#import <Cocoa/Cocoa.h>
#import <QTKit/QTKit.h>
```

2. Add the following declaration code after your import statements:

```
@interface OpenURLPanel : NSObject
{
// panel
 IBOutlet NSPanel   *mPanel;
 IBOutlet NSComboBox *mUrlComboBox;
// open url panel
    id              mDelegate;
    SEL             mDidEndSelector;
    void            *mContextInfo;
    NSMutableArray  *mUrlArray;
```

```
    BOOL            mIsSheet;
}
```

3. Define a class method with the following line of code:

```
+ (id)openURLPanel;
```

4. Define the getters you need with the following lines of code:

```
-(NSString *)urlString;
- (NSURL *)url;
```

5. Define the setters with the following line of code:

```
- (void)setURLArray:(NSMutableArray *)urlArray;
```

6. Define the delegate with the following line of code:

```
- (void)awakeFromNib;
```

7. Define the notifications:

```
- (void)writeURLs:(NSNotification *)notification;
```

8. Define the actions you need:

```
(IBAction)doOpenURL:(id)sender;
```

9. Define the delegate methods:

```
 - (void)beginSheetWithWindow:(NSWindow *)window delegate:(id)delegate
didEndSelector:(SEL)didEndSelector contextInfo:(void *)contextInfo;
```

Save the file in the Classes folder in your QTKitPlayer project.

Adding Code To OpenURLPanel.m

In this next sequence of steps, you'll be adding a larger chunk of code to your `OpenURLPanel.m` implementation file.

To begin, in your QTKitPlayer project, choose File > New File. In the Assistant window for your new file in Xcode 2.0, select Cocoa > Objective-C class and in the window that opens enter the title `OpenURLPanel.m`. (Note that if you check the box in the title window, your implementation will already be created for you.) Now follow these steps:

1. Insert the following import code at the beginning of your file:

```
#import "OpenURLPanel.h"
```

2. Following your import statement, you want to define a constant for specifying user default keys. Insert this line:

```
#define kUserDefaultURLsKey @"UserDefaultURLsKey"
```

3. You also want to define the maximum number of URLs, in this case 15. Insert this line:

```
#define kMaximumURLs 15
```

4. Now you want to add the following class methods to deal with opening the URL panel. Add these lines:

```
+ (id)openURLPanel
{
 if (openURLPanel == nil)
 openURLPanel = [[self alloc] init];
return openURLPanel;
}
```

5. Next, you need to add initialization code to initialize an OpenURLPanel instance and listen for application termination notifications. Add these lines:

```
- (id)init
{
    [super init];

    // init
    [self setURLArray:[NSMutableArray arrayWithCapacity:10];

    [[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(writeURLs:) name:NSApplicationWillTerminateNotification
object:NSApp];

    return self;
}
```

6. Insert the following code to handle deallocation of memory and notifications:

```
- (void)dealloc
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
    [self setURLArray:nil];
    [super dealloc];
}
```

7. To get the URL string, insert this chunk of code:

```
- (NSString *)urlString
{
     NSString *urlString = nil;

    // get the url
     urlString = [mUrlComboBox stringValue];

    if (urlString == nil)
        urlString = [mUrlComboBox objectValueOfSelectedItem];

    if ([urlString length] == 0)
        urlString = nil;

    return urlString;
}
```

8. To set the instance variable of the URL array, add the following code:

```
- (void)setURLArray:(NSMutableArray *)urlLArray
{
```

```
    [urlLArray retain];
    [mUrlLArray retain];
     mUrlLArray = urlLArray;
}
```

9. The next block of code lets you restore the previous URLs. Insert the following block of code:

```
- (void)awakeFromNib
{
    NSArray *urls;

    // restore the previous urls
    urls = [[NSUserDefaults standardUserDefaults]
objectForKey:kUserDefaultURLsKey];
 [mUrlArray addObjectsFromArray:urls];

    if (urls)
 [mUrlComboBox addItemsWithObjectValues:urls];
}
```

10. To set up deal with notifications, you need to add this chunk of code. This will enable you to "listen" for any notifications. Insert the following:

```
- (void)writeURLs:(NSNotification *)notification
{

    NSUserDefaults *userDefaults;
    if ([mUrlArray count]
    {
    // init
 userDefaults = [NSUserDefaults standardUserDefaults];

    // write out the urls
    [userDefaults setObject:mUrlArray forKey:kUserDefaultURLsKey];
    [userDefaults synchronize];
        }
}
```

11. To write to actions, validate and save the URL, add the following code:

```
- (IBAction)doOpenURL:(id)sender
{
{
    NSString*urlString;
    NSURL*url;
    BOOLinformDelegate = YES;
    IMP     callback;

    if ([sender tag] == NSOKButton)
    {
        // validate the URL
        url = [self url];
        urlString = [self urlString];

        if (url)
        {
            // save the url
            if (![mUrlArray containsObject:urlString])
```

```
            {
                // save the url
                [mUrlArray addObject:urlString];

                // add the url to the combo box
                [mUrlComboBox addItemWithObjectValue:urlString];

                // remove the oldest url if the maximum has been exceeded
                if ([mUrlArray count] > kMaximumURLs)
                {
                    [mUrlArray removeObjectAtIndex:0];
                    [mUrlComboBox removeItemAtIndex:0];
                }
            }
            else
            {
                // move the url to the bottom of the list
                [mUrlArray removeObject:urlString];
                [mUrlArray addObject:urlString];
                [mUrlComboBox removeItemWithObjectValue:urlString];
                [mUrlComboBox addItemWithObjectValue:urlString];
            }
        }
        else
        {
            if (mIsSheet)
                NSRunAlertPanel(@"Invalid URL", @"The URL is not valid.", nil,
 nil, nil);
            else
                NSBeginAlertSheet(@"Invalid URL", nil, nil, nil, mPanel, nil,
nil, nil, nil, @"The URL is not valid.");

            informDelegate = NO;
        }
    }

    // inform the delegate
    if (informDelegate && mDelegate && mDidEndSelector)
    {
        callback = [mDelegate methodForSelector:mDidEndSelector];
        callback(mDelegate, mDidEndSelector, self, [sender tag], mContextInfo);
    }
}
```

12. Add these lines of code to save the delegate and start the sheet:

```
- (void)beginSheetWithWindow:(NSWindow *)window delegate:(id)delegate
didEndSelector:(SEL)didEndSelector contextInfo:(void *)contextInfo
{
    // will this run as a sheet
    mIsSheet = (window ? YES : NO);

    // save the delegate, did end selector, and context info
    mDelegate = delegate;
    mDidEndSelector = didEndSelector;
    mContextInfo = contextInfo;

    // load the bundle (if necessary)
```

```
    if (mPanel == nil)
        [NSBundle loadNibNamed:@"OpenURLPanel" owner:self];

    // start the sheet (or window)
    [NSApp beginSheet:mPanel modalForWindow:window modalDelegate:nil
didEndSelector:nil contextInfo:nil];
}
```

13. Add these lines to close it down:

```
- (void)close
{
    // close it down
    [NSApp endSheet:mPanel];
    [mPanel close];
}
```

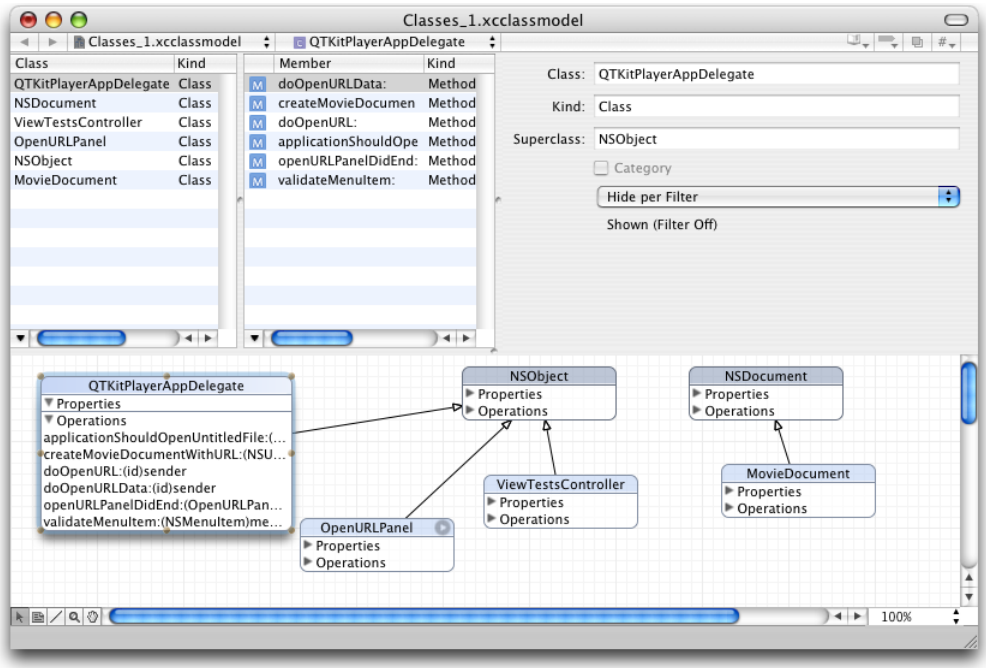
This completes the steps for adding code to your `OpenURLPanel.m` implementation file. There is only one more sequence of steps, described in the next section, before you can run and build your QTKitPlayer application for streaming audio and video.

Adding Code to QTKitPlayerDelegate

In this next sequence of steps, you'll be adding a small amount of code to your `QTKitPlayerAppDelegate.h` class interface file.

In Xcode 2.0, you'll see the class model for the delegate, as shown in Figure 5-12.

Figure 5-12 Class model for delegate



To begin, in your QTKitPlayer project, choose File > New File. In the Assistant window for your new file in Xcode 2.0, select Cocoa > Objective-C class and in the window that opens enter the title `QTKitPlayerAppDelegate.h` and check the box that also lets you create a `QTKitPlayerAppDelegate.m` file. Now follow these steps:

1. Insert the following import code at the beginning of your file:

```
#import <Cocoa/Cocoa.h>
#import "OpenURLPanel.h"
```

2. Add the following declaration code after your import statements:

```
@interface QTKitPlayerAppDelegate : NSObject
```

3. Define a NSMenu validation protocol with the following line of code:

```
- (BOOL)validateMenuItem:(NSMenuItem *)menuItem;
```

4. Define the OpenURLPanel delegates with the following lines of code:

```
- (void)openURLPanelDidEnd:(OpenURLPanel *)openURLPanel returnCode:(int)returnCode
 contextInfo:(void *)contextInfo;
```

5. Define the actions with the following lines of code:

```
- (IBAction)doOpenURL:(id)sender;
- (IBAction)doOpenURLData:(id)sender;
```

6. Define the method with the following line of code:

```
 - (BOOL)createMovieDocumentWithURL:(NSURL *)url asData:(BOOL)asData;
```

You're done with the `QTKitPlayerAppDelegate.h` declaration file.

Adding Code To Your QTKitPlayerDelegate.m

In this next sequence of steps, you'll be adding a larger chunk of code to your `QTKitPlayerAppDelegate.m` implementation file.

To begin, in the `QTKitPlayerAppDelegate.m` file, you want to follow these steps:

1. Insert the following import code at the beginning of your file:

```
#import "QTKit/QTKit.h"
#import "QTKitPlayerAppDelegate.h"
#import “MovieDocument.h.”
```

2. Following your import statement, you want add these enumerations. Insert these lines:

```
{
    kQTKitPlayerOpenAsURL = 0,
    kQTKitPlayerOpenAsData

};
```

3. You also want to add `QTKitPlayAppDelegate` after the `@implementation` directive. Insert this line:

```
@implementation QTKitPlayerAppDelegate
```

4. Now you want to add the following lines for NSMenu validation protocols. Insert this block of code:

```
- (BOOL)validateMenuItem:(NSMenuItem *)menuItem
{
    BOOLvalid = NO;
    SEL action;

    // init
    action = [menuItem action];

    // validate
    if (action == @selector(doOpenURL:))
        valid = YES;
    else if (action == @selector(doOpenURLData:))
        valid = YES;
    else
        valid = [[NSDocumentController sharedDocumentController]
validateMenuItem:menuItem];

    return valid;
}
```

5. Next, you need to add these OpenURLPanel delegates. Insert these lines:

```
- (void)openURLPanelDidEnd:(OpenURLPanel *)openURLPanel returnCode:(int)returnCode
 contextInfo:(void *)contextInfo
{
    BOOL closePanel = YES;

    // create the movie document
    if (returnCode == NSOKButton)
        closePanel = [self createMovieDocumentWithURL:[openURLPanel url]
asData:((long)contextInfo == kQTKitPlayerOpenAsData)];

    if (closePanel)
        [openURLPanel close];
}
```

6. Insert the following code to handle the necessary actions for opening the sheet with a window:

```
- (IBAction)doOpenURL:(id)sender
{
    [[OpenURLPanel openURLPanel] beginSheetWithWindow:nil delegate:self
didEndSelector:@selector(openURLPanelDidEnd:returnCode:contextInfo:)
contextInfo:((void *)kQTKitPlayerOpenAsURL)];
}

- (IBAction)doOpenURLData:(id)sender
{
    [[OpenURLPanel openURLPanel] beginSheetWithWindow:nil delegate:self
didEndSelector:@selector(openURLPanelDidEnd:returnCode:contextInfo:)
contextInfo:((void *)kQTKitPlayerOpenAsData)];
}
```

7. These are the methods you need to create and set up the movie document with an associated URL. Insert this chunk of code:

```
- (BOOL)createMovieDocumentWithURL:(NSURL *)url asData:(BOOL)asData
{
    NSDocument  *movieDocument = nil;
    NSDocumentController*documentController;
    BOOL            success = YES;

    // init
    documentController = [NSDocumentController sharedDocumentController];

    // try to create the document from the URL
    if (url)
    {
        if (asData)
          movieDocument = [documentController makeDocumentWithContentsOfURL:url
 ofType:@"MovieDocumentData"];
        else
          movieDocument = [documentController makeDocumentWithContentsOfURL:url
 ofType:@"MovieDocument"];
    }

    // add the document
    if (movieDocument)
    {
        [documentController addDocument:movieDocument];

        // setup
        [movieDocument makeWindowControllers];
        [movieDocument updateChangeCount:NSChangeCleared];
        [movieDocument showWindows];
    }
    else
    {
       NSRunAlertPanel(@"Invalid movie", @"The url is not a valid movie.", nil,
 nil, nil);
        success = NO;
    }

    return success;
}
```

This wraps up the code additions for your QTKitPlayer application. Now you're ready to build and compile the application and play streaming audio and video over the Internet.

What's Next?

You've extended the QTKitPlayer beyond its early incarnations as a simple media player, adding editing, importing, exporting, and now streaming capabilities. In the next chapter, you'll provide new enhancements to the player that will enable you to play back as many as six different QuickTime movies, including QuickTime VR, streaming audio and video, wired sprite movies, and other multimedia content.

You'll have a multimedia engine at your disposal that you can use to launch a variety of QuickTime content, much as you would do with an interactive movie kiosk. The steps to create this engine are spelled out in the next chapter and enable you to build on what you've learned so far in this programming guide.

Adding Multimedia Playback Capability

In this chapter, you'll add multimedia playback to your QTKitPlayer, working with the tools available to you in Interface Builder and Xcode 2.0. The goal is, as in previous chapters, to build on the code you've written and the interface you've constructed, so that you can enhance the capabilities of the QTKitPlayer—with a minimum of programming effort.

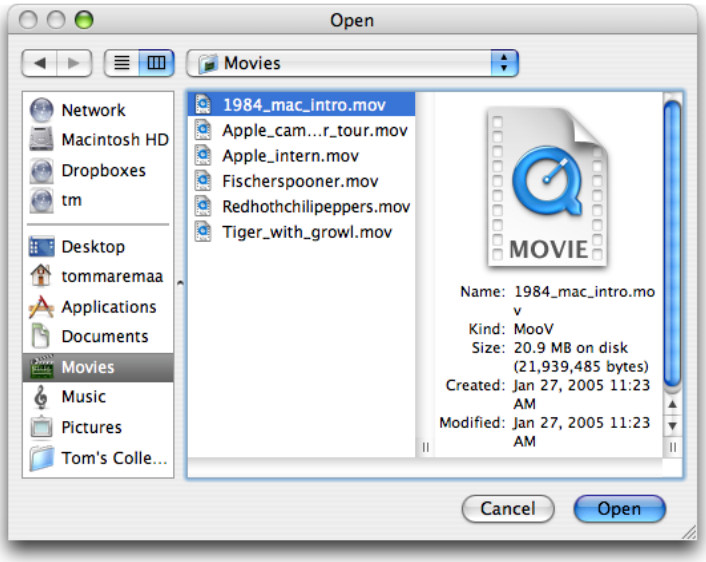
When completed, the results will be dramatic and visually exciting: a multimedia playback engine capable of simultaneously playing in real time as many as six different QuickTime movies, QuickTime VR panoramas and object movies, streaming audio and video, animation and wired sprite movies, and other content that QuickTime can import and display. The user experience will be enhanced and require some degree of interactivity with the multimedia content that is displayed. Just controlling multichannel sound or interactive VR movies will alter the user experience.

Note: Multimedia is an often misused term, with different meanings, implied or otherwise. To developers of digital media, "multimedia is defined by a set of standards that enable media to be acquired, represented, compressed, delivered and displayed," according to Phillip John McKerrow, a computer scientist and teacher of multimedia programming, in a recent article in the journal *IEEE Multimedia*. "For a production to be considered multimedia it must include a provision for the user to interact with the content and influence the course of the presentation."

In the completed project, you'll add a new menu title, Studio, to the QTKitPlayer and a menu item, Present Movies. You'll also add code to open and display all six QuickTime movies in a window, with multiple—and resizable—views of each movie.

Clicking a button in the Play Multimedia Content window will open a dialog box from which users will be able to select any movie or media type of their choice (Figure 6-1). After each movie is chosen, a new dialog box appears, prompting the user for another selection until the user has populated the window with all six QuickTime movies, as shown in Figure 6-2.

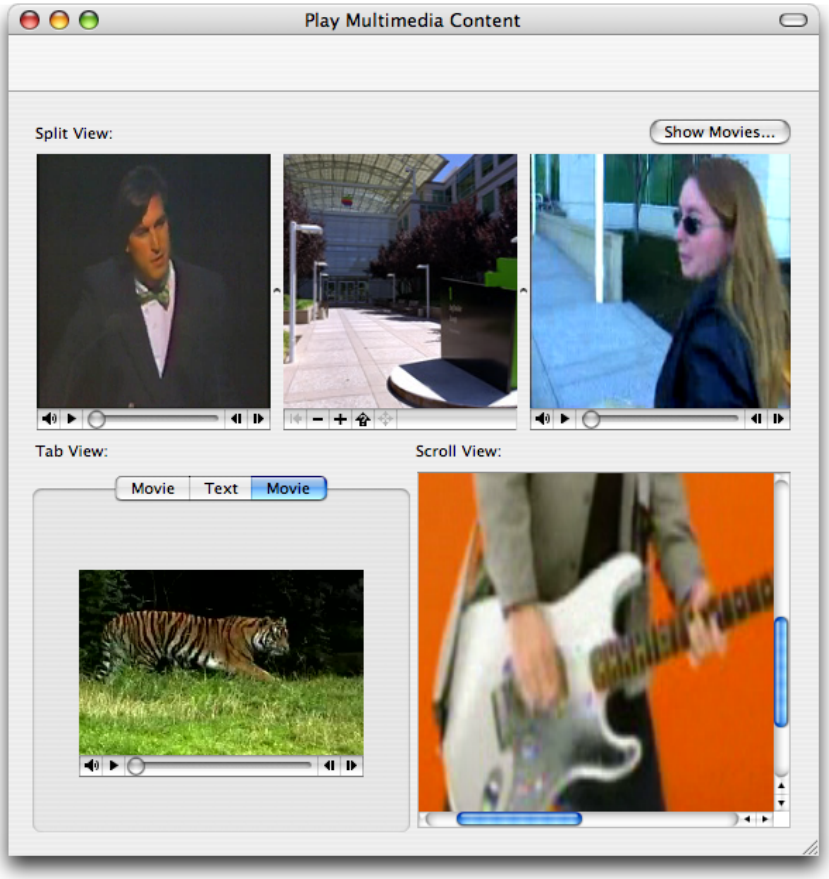
Figure 6-1 Opening six QuickTime movies of the user's choosing for display in the multimedia content window of the QTKitPlayer application



The window in which the movies are displayed can be resized to full screen for maximum visual impact and the toolbar at the top can be collapsed for a kiosk-like effect. The three movies displayed at the top of window contain an NSSplitView, while an NSTabView is provided for two movies and a text view below, and an NSScrollView with a movie appears in the right corner of the window.

The intended effect is for some degree of user interactivity with one or all of the QuickTime movies displayed, either by splitting the views and tabbing through them, or starting and stopping the playback of each movie. If all six movies are different QuickTime VR panoramas or object movies, for example, the user will be able to point and click through a variety of landscapes or hotspots from different points of view. The result is a heightened user experience.

Figure 6-2 All six movies selected by the user playing in different views



Users may interact as they would with a multimedia kiosk or presentation, playing each movie at a different rate using the playback commands available in the QTKitPlayer.

At the programming level, each movie is simply a QTMovieView object and is intended to interact with other standard Cocoa views. The Show Movies button in the Play Multimedia Content window will launch each of the six dialogs that enable the user to choose a movie for display and playback. Users can then control the playback of each movie using the commands available in the QTKitPlayer to start, stop, go to the beginning, go to the end, show poster frame, step forward and step backward. Movies can also be controlled by toggling the spacebar to start and stop playback. Movie editing, however, is not enabled for any movie being displayed.

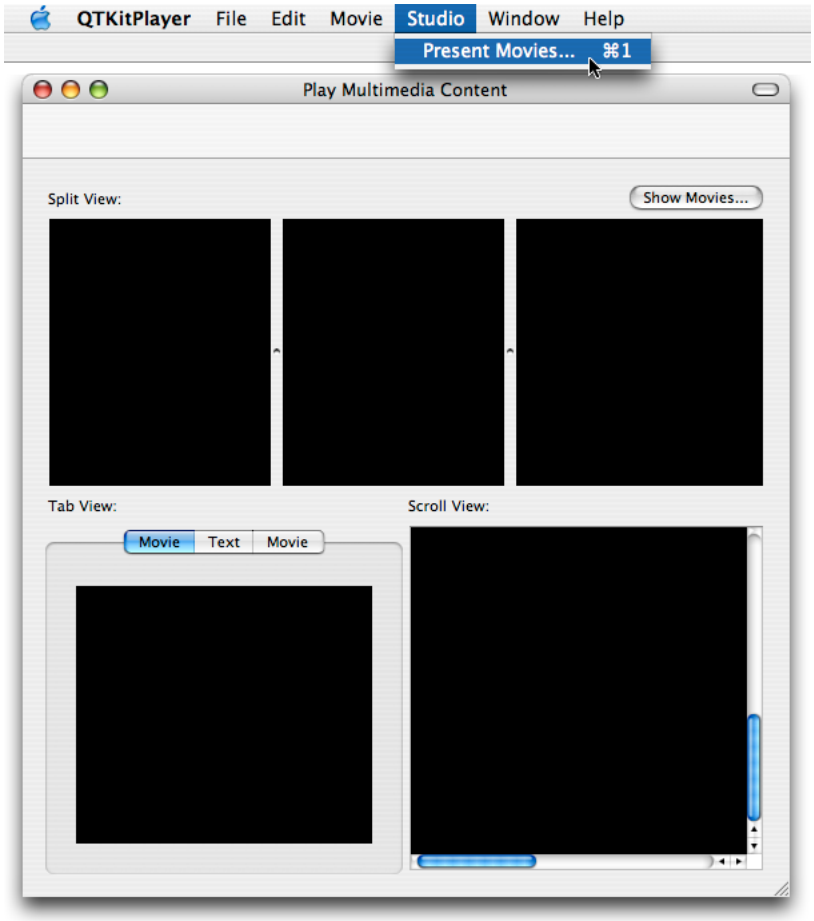
If you've worked through the examples in the previous chapters, you'll be ready to move ahead with constructing and coding the multimedia playback enhancement for your QTKitPlayer.

Tasks to Accomplish

The tasks you want to accomplish in adding this new functionality to your QTKitPlayer application are less complex than those in the previous chapter. You'll build on the existing QTKitPlayer, using the tools available to you in Interface Builder 2.5 and Xcode 2.0. The code you'll need to add to the project in order to make it work will be surprisingly simple—but powerful. You'll do this:

1. Add a View Tests Window to your MainMenu.nib in Interface Builder and populate that window with the QTMovieView objects you need to display QuickTime movies, as indicated in Figure 6-3.
2. Specify the attributes of the View Tests Window you've added to the nib.
3. Subclass NSObject with a `ViewTestsController` class in your MainMenu.nib and wire it up with outlets and actions to handle the opening and display of movies in the View Tests Window.
4. Add to your Xcode project a new `ViewTestsController.h` declaration file in which you define the instance variables and actions for your `ViewTestsController` class.
5. Add an `ViewTestsController.m` implementation file to your QTKitPlayer project in which you handle getting and setting the movies you want to play, notifications, stopping and clearing the movies that are playing, and adding a toolbar which can be shown or not shown.
6. Add a new menu title to the QTKitPlayer and a menu item to open the window
7. Connect the Show Movies button to the ViewTestsController object.

Figure 6-3 The layout of the objects in the content window with the Present Movies menu item selected



Each task is outlined in the next sections of this chapter. You'll start as you have before with Interface Builder and then move on to add the code you need in two separate Xcode files.

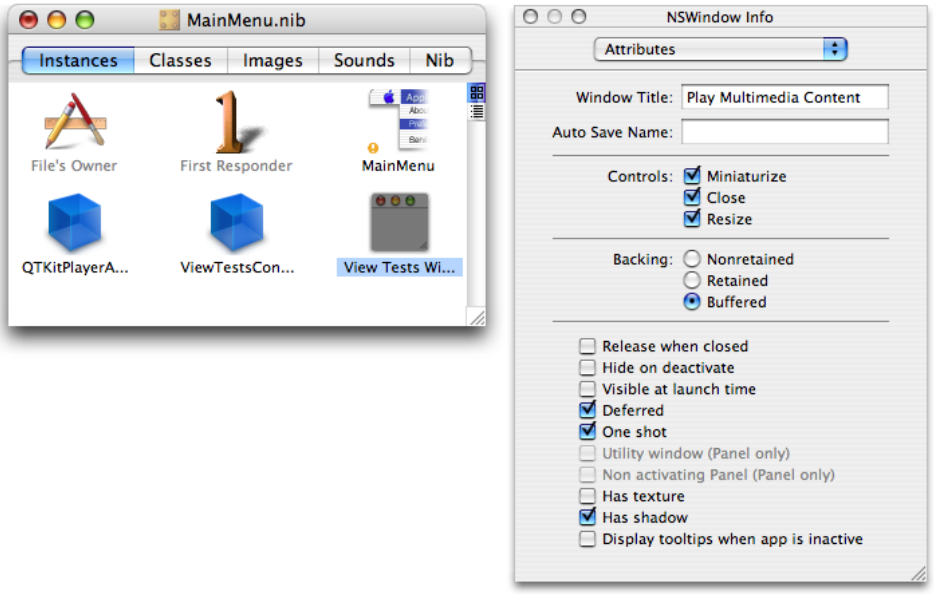
Constructing The Multimedia Playback Engine

By now you should be familiar with how to work with Interface Builder and its various palettes, icons, and objects. For purposes of simplicity and to move things forward a bit faster, this means combining a few basic steps in constructing your multimedia playback engine with all the different QTMovieView objects in view. To start:

1. Drag a window object from the Cocoa-Windows palette in Interface Builder into your MainMenu.nib and name it View Tests Window.

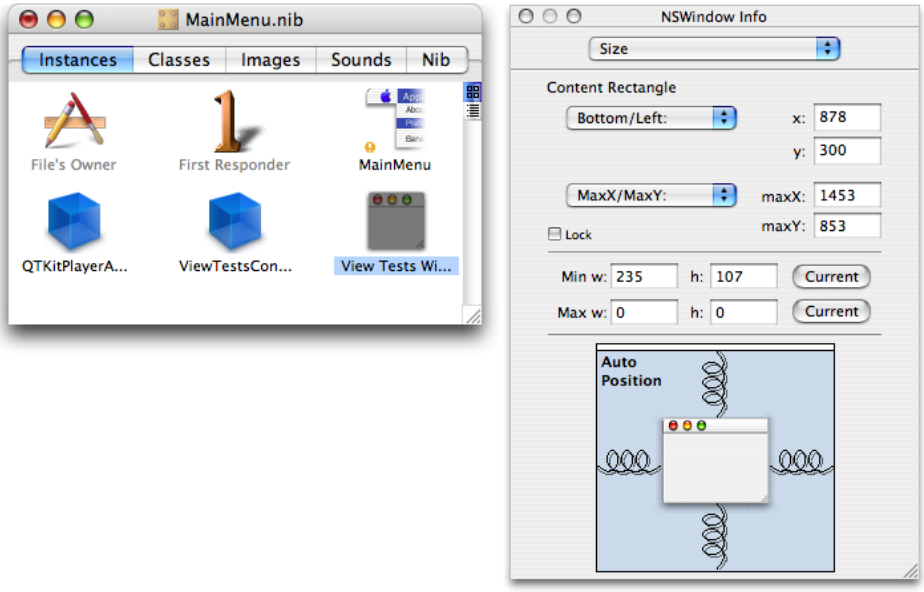
2. Command-1 to open the NSWindow info panel and set the attributes for the window object, as shown in Figure 6-4. Enter the window title Play Multimedia Content.

Figure 6-4 The attributes pane for the View Tests Window object



3. Press Command-3 to open the Size of the View Tests Window and set the size and springs, as shown in Figure 6-5.

Figure 6-5 The size of the View Tests Window with the springs set



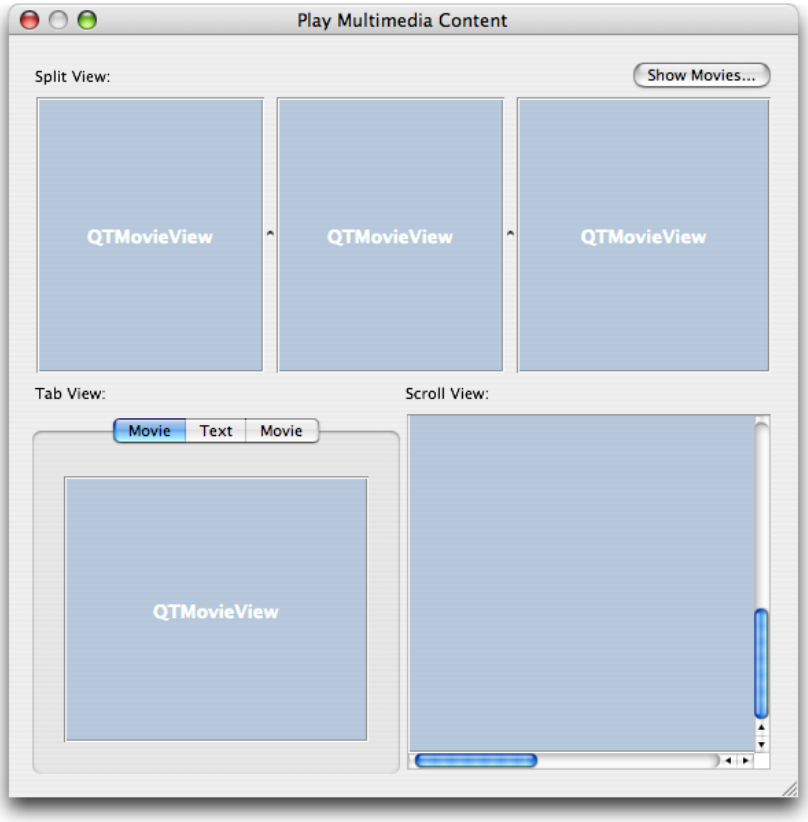
4. Now you want to construct your Play Multimedia Content window, as shown completed in Figure 6-6. Drag CustomView objects into the window and subclass them as QTMovieView objects. (In the Info window, use the Custom Class pop-up to subclass the objects.)

5. Figure 6-6 shows a lot of steps that may not be obvious. To simplify, do this: Drag the TabView object into the window from the palette. Drag three CustomView objects into the window, select all three by shift-clicking, and choose Layout > Make Subviews Of > Split View. Repeat this for the ScrollView. This is the preferred way of working with these objects in your window.

6. Now you want to drag CustomView objects onto the TabView, SplitView, and ScrollView objects. Subclass them as QTMovieView objects.

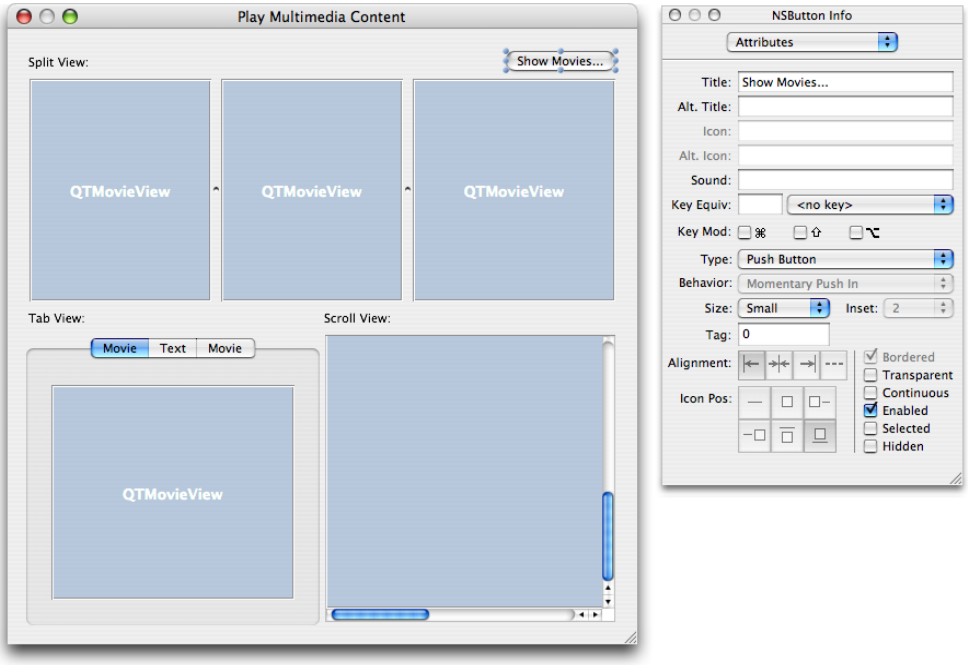
7. Add textfields for Split View:, Tab View:, and Scroll View. Also, you want to add buttons in Tab View for Movie, Text, and Movie, as shown in Figure 6-6.

Figure 6-6 The layout of QTMovieView objects with textfields added for different views



8. Drag a button from the Cocoa-Controls palette into the upper right corner of the window. Name the button Show Movies and specify its attributes, as shown in Figure 6-7.

Figure 6-7 Button attributes

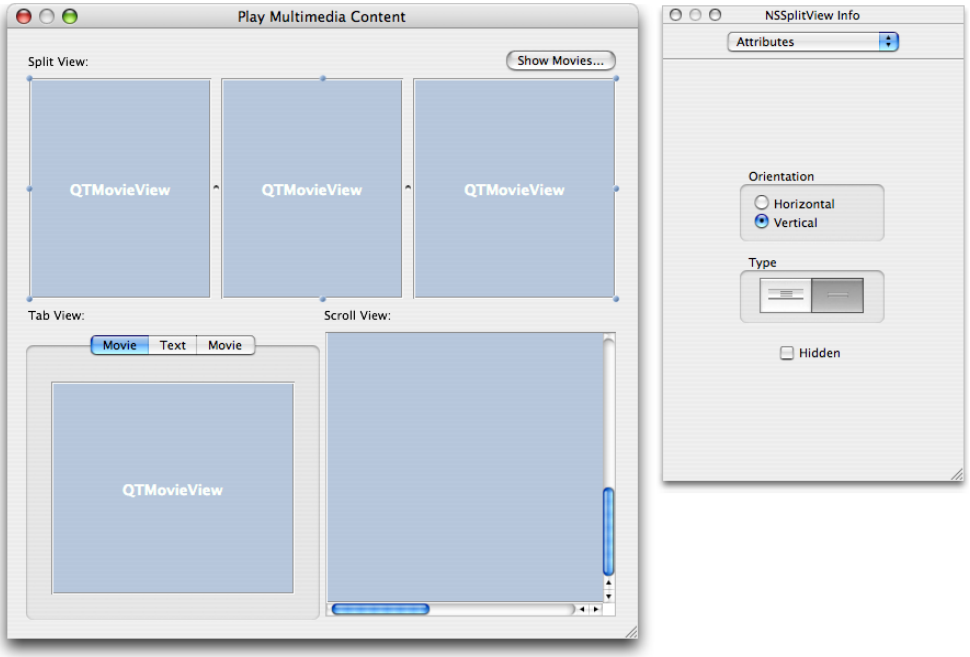


9. Create an action `doSetMovies:` for the Show Movies button in the upper right hand corner of the window. Double-click the ViewTestsController object. Click the actions pane. Click the Add button to add an action `doSetMovies:`.

10. Wire the Show Movies button to the doSetMovies: action. Click t he Show Movies button and press the Control key. Click-drag from the button to the ViewTestsController object. In the Info window for the ViewTestsController object, click the Connect button.

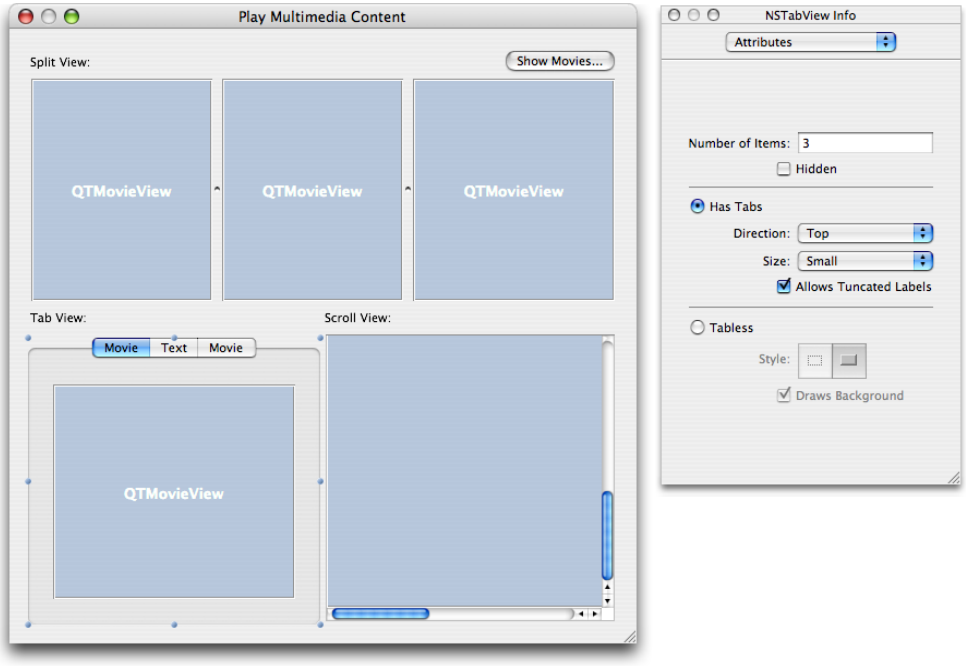
11. Now you want to specify the window attributes for the Split View, shown in Figure 6-8.

Figure 6-8 Split View attributes specified



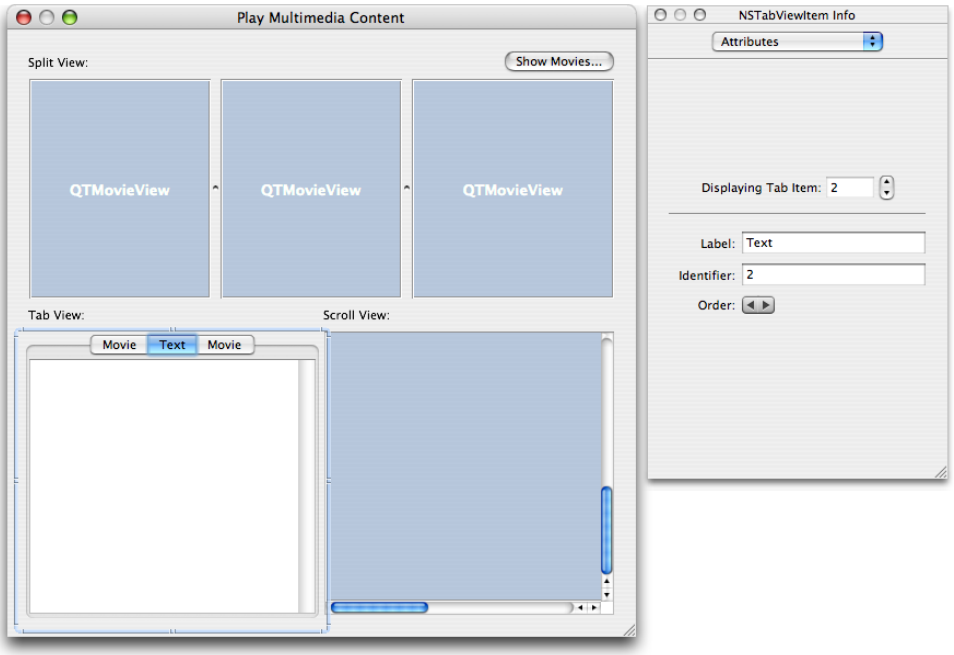
12. Specify the window attributes for the Tab View. Note the number of items in the TabView object, as shown in Figure 6-9.

Figure 6-9 Tab view attributes specified



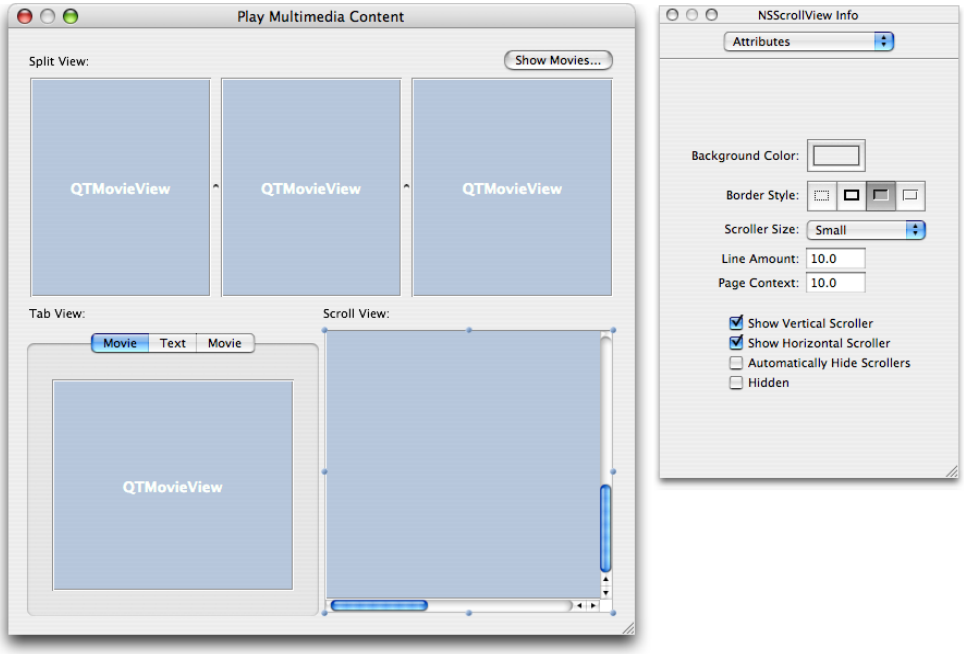
13. Specify the attributes in the Text TabViewItem, as shown in Figure 6-10. The user will be able to enter text into this window.

Figure 6-10 Text attribute specified



14. Specify the attributes of the Scroll View object, with border style, and vertical and horizontal scroller checked, as shown in Figure 6-11.

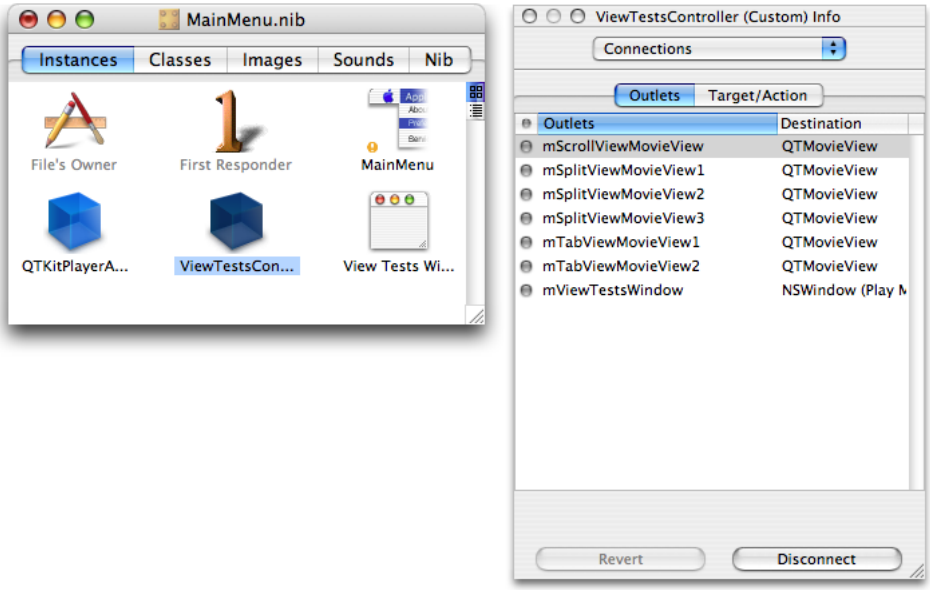
Figure 6-11 Scroll View attributes specified



15. To make the controller, subclass NSObject, and name the controller ViewTestsController.

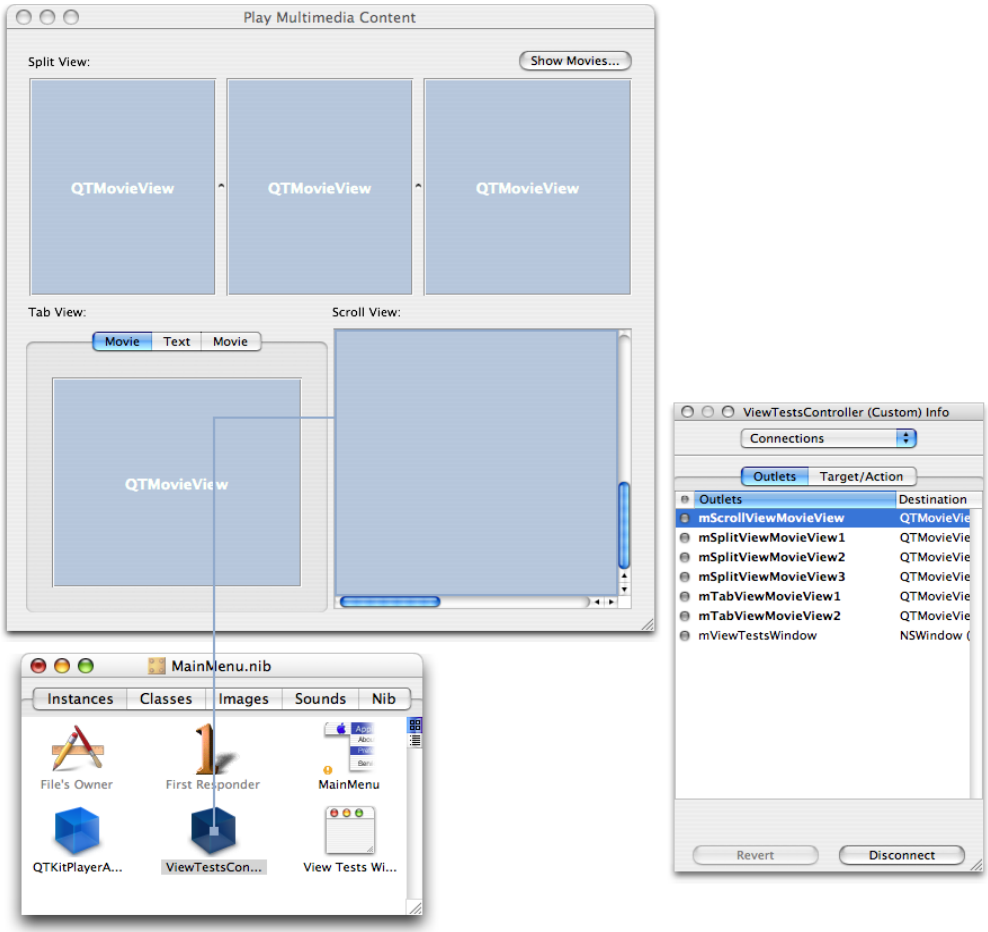
16. Add outlets to ViewTestsController. Double-click the ViewTestsController object. In the Info window, click the Add button to add the following outlets: `mScrollViewMovieView`, `mSplitViewMovieView1`, `mSplitViewMovieView2`, `mSplitViewMovieView3`, `mTabViewMovieView1`, `mTabViewMovieView2`, and `mViewTestsWindow`, as shown in Figure 6-12.

Figure 6-12 Outlet connections for the ViewTestsController



17. Now you want to wire up the connections from the various QTMovieView objects in the window to their respective outlets. Control-drag from the ViewTestController object to the ScrollView, TabView, and SplitView objects in the window and click the Connect button, as shown in Figure 6-13.

Figure 6-13 Connecting the ViewTestsController to an outlet

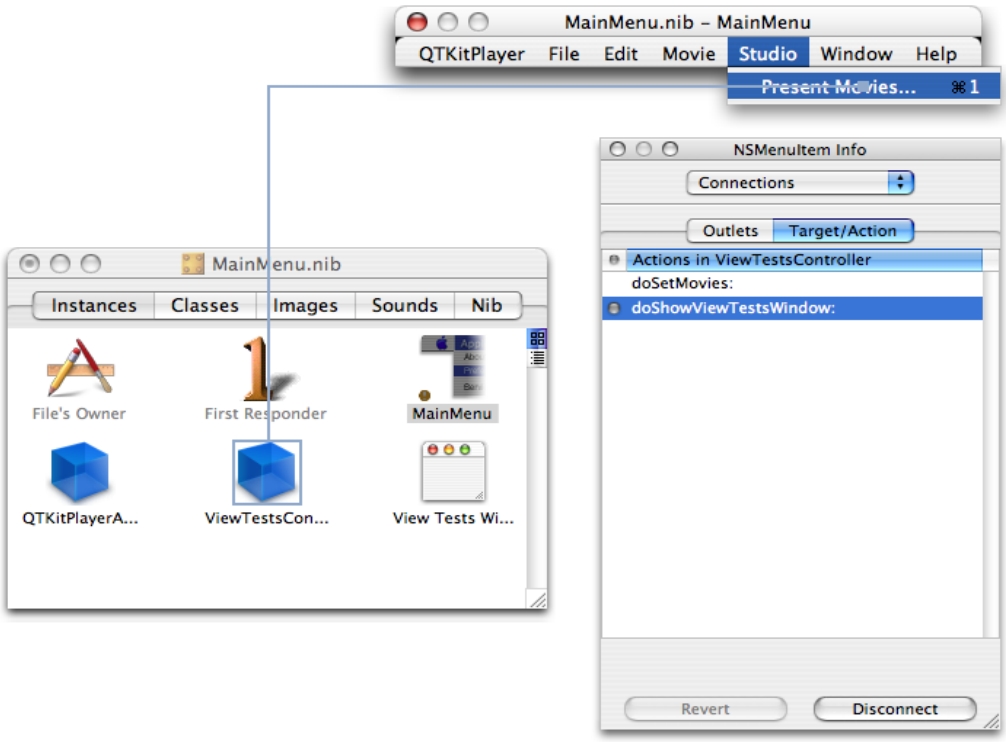


18. Add another menu to the MainMenu.nib - MainMenu in Interface Builder with the title Studio, and add a menu item entitled Present Movies with a Command-1 keystroke equivalent.

19. Add an action `doShowViewTestsWindow:` to ViewTestsController. Double-click the ViewTestsController icon. In the Info window, click the Actions pane. Click the Add button to add an action `doShowViewTestsWindow:`.

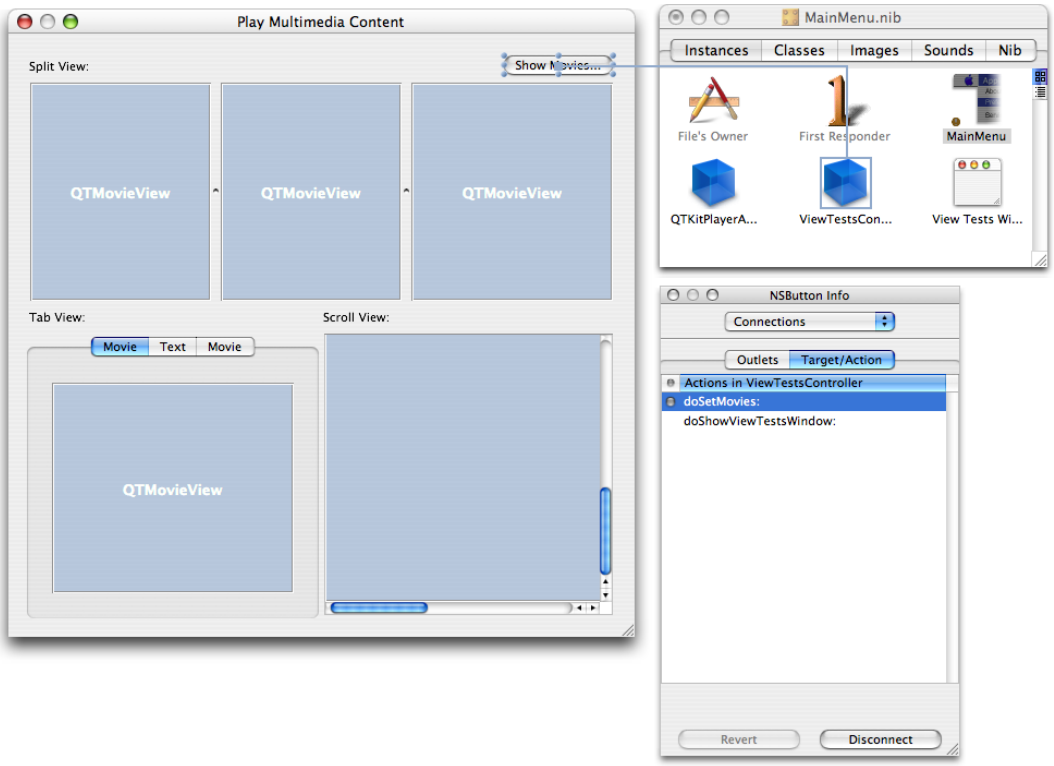
20. Connect the Present Movies menu item by pressing the Control key and drag-connecting the wire to the `ViewTestsController` object. Make the connection to the `doShowViewTestsWindow` target, as shown in Figure 6-14.

Figure 6-14 The present movie connection to the ViewTestsController and target



21. Connect the Show Movies button to the ViewTestsController object, as shown in Figure 6-15.

Figure 6-15 Connecting the Show Movies button to the ViewTestsController with its target

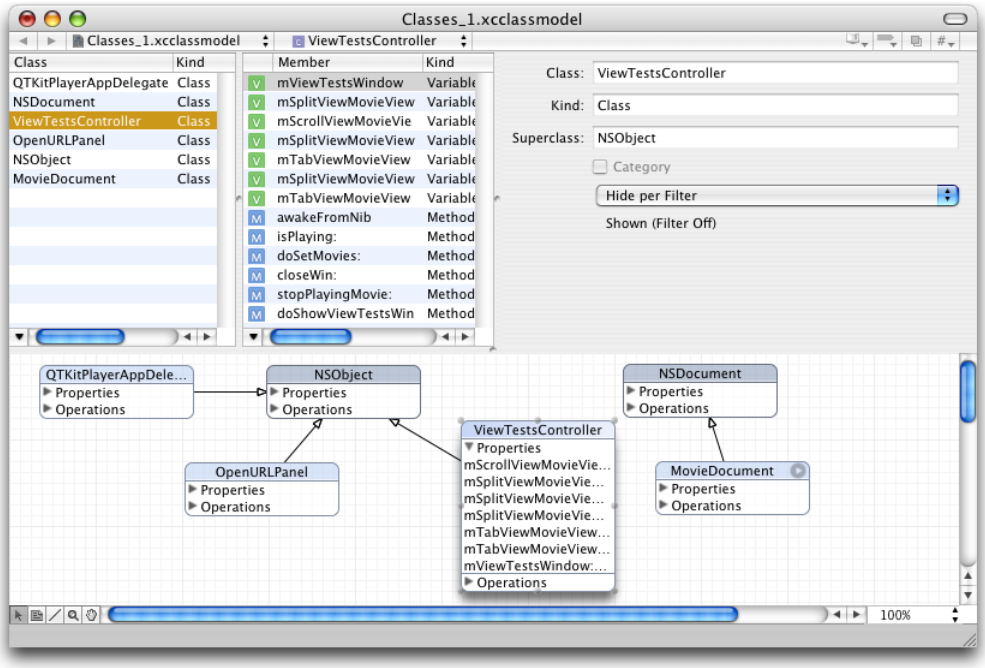


This completes the steps you need to follow in Interface Builder. You're now ready to add the code to make the project work.

Adding Code To Display and Playback Multimedia

In this section, you'll add two new files to your Xcode project, including a `ViewTestsController.h` declaration file and an `ViewTestsController.m` implementation file. Figure 6-16 shows the class model for the `ViewTestsController` class with its properties and their connections listed

Figure 6-16 The class model in Xcode 2.0 of the ViewTestsController class



Adding Code To Your ViewTestsController Class Interface

In this next sequence of steps, you'll be adding a small amount of code to your `ViewTestsController.h` class interface file.

To begin, in your QTKitPlayer project, choose File > New File. In the Assistant window for your new file in Xcode 2.0, select Cocoa > Objective-C class and in the window that opens enter the title `ViewTestsController.h`. Now follow these steps:

1. Insert the following import code at the beginning of your file:

```
#import <Cocoa/Cocoa.h>
#import <QTKit/QTKit.h>
```

2. Add this line of code:

```
@class QTMovieView;
```

3. Add the following block of code declaring the instance variables that belong to your `ViewTestsController` class:

```
@interface ViewTestsController : NSObject
{
    IBOutlet NSWindow    *mViewTestsWindow;
    IBOutlet QTMovieView *mSplitViewMovieView1;
    IBOutlet QTMovieView *mSplitViewMovieView2;
    IBOutlet QTMovieView *mSplitViewMovieView3;
    IBOutlet QTMovieView *mTabViewMovieView1;
    IBOutlet QTMovieView *mTabViewMovieView2;
    IBOutlet QTMovieView *mScrollViewMovieView;
```

```
}
```

4. Define the actions you need before the `@end` directive in order to show movies in your window and to show the window itself:

```
- (IBAction)doSetMovies:(id)sender;
- (IBAction)doShowViewTestsWindow:(id)sender;
```

That's it. Save your `ViewTestsController.h` file.

Adding Code To Your ViewTestController.m

In this next sequence of steps, you'll be adding a larger chunk of code to your `ViewTestsController.m` implementation file.

To begin, in your QTKitPlayer project, choose File > New File. In the Assistant window for your new file in Xcode 2.0, select Cocoa > Objective-C class and in the window that opens enter the title `ViewTestsController.m`. Now follow these steps

1. Insert the following import line at the beginning of your file:

```
#import "ViewTestsController.h"
```

2. Following your import statement, you want to add this line:

```
@implementation ViewTestsController
```

3. You want to be notified when the window is closing so you can do a cleanup. Insert these lines of code:

```
-(void)awakeFromNib
{
 [ [NSNotificationCenter defaultCenter]
    addObserver:self selector:@selector(closeWin:)
 name:NSWindowWillCloseNotification object:mViewTestsWindow ];
}
```

4. Add this next block to specify if the movie is playing or not. If the movie is currently playing, it returns `YES`. If not, it returns `NO`.

```
-(BOOL)isPlaying:(QTMovie *)aMovie
{
 if ([aMovie rate] == 0)
 {
return NO;
}
return YES:
}
```

5. Next, you need to add code to stop the movie from playing. Add these lines:

```
-(void)stopPlayingMovie:(QTMovie *)aMovie
{
    if ([self isPlaying:aMovie] == YES)
    {
        [aMovie stop];
```

```
    }
}
```

6. Insert the following code to handle stopping of any currently playing movies and to clear the movies that were previously set for each view:

```
- (void)closeWin:(void *)userInfo
{
    // stop any currently playing movies
    [self stopPlayingMovie:[mSplitViewMovieView1 movie]];
    [self stopPlayingMovie:[mSplitViewMovieView2 movie]];
    [self stopPlayingMovie:[mSplitViewMovieView3 movie]];
    [self stopPlayingMovie:[mTabViewMovieView1 movie]];
    [self stopPlayingMovie:[mTabViewMovieView2 movie]];
    [self stopPlayingMovie:[mScrollViewMovieView movie]];

    // clear the movies that were previously set for each view
    [mSplitViewMovieView1 setMovie:NULL];
    [mSplitViewMovieView2 setMovie:NULL];
    [mSplitViewMovieView3 setMovie:NULL];
    [mTabViewMovieView1 setMovie:NULL];
    [mTabViewMovieView2 setMovie:NULL];
    [mScrollViewMovieView setMovie:NULL];

}
```

7. To get the movies you want to play and open the panel so the user can choose, insert this chunk of code:

```
- (QTMovie *)getAMovieFile
{
    NSOpenPanel *openPanel;
 openPanel = [NSOpenPanel openPanel];
    [openPanel setCanChooseDirectories:NO];

    if ([openPanel runModalForTypes:[QTMovie movieUnfilteredFileTypes]] ==
NSOKButton)
    {
        return [QTMovie movieWithFile:[openPanel filename] error:NULL];
    }

    return nil;
```

8. To set the movies, add this chunk of code:

```
- (IBAction)doSetMovies:(id)sender
    // set the movies
 [mSplitViewMovieView1 setMovie:[self getAMovieFile]];
    [mSplitViewMovieView2 setMovie:[self getAMovieFile]];
    [mSplitViewMovieView3 setMovie:[self getAMovieFile]];
    [mTabViewMovieView1 setMovie:[self getAMovieFile]];
    [mTabViewMovieView2 setMovie:[self getAMovieFile]];
    [mScrollViewMovieView setMovie:[self getAMovieFile]];
}
```

9. To add a toolbar and to show the window, add the following code:

```
- (IBAction)doShowViewTestsWindow:(id)sender
{
```

```
    // add a toolbar
    if ([mViewTestsWindow toolbar] == nil)
        [mViewTestsWindow setToolbar:[[[NSToolbar alloc]
initWithIdentifier:@"QTKitPlayer"] autorelease]];

    // show the window
    [mViewTestsWindow makeKeyAndOrderFront:nil];
}

@end
```

This completes the steps for adding code to your `ViewTestsController.m` implementation file. You can build and compile your QTKitPlayer application for multimedia playback.

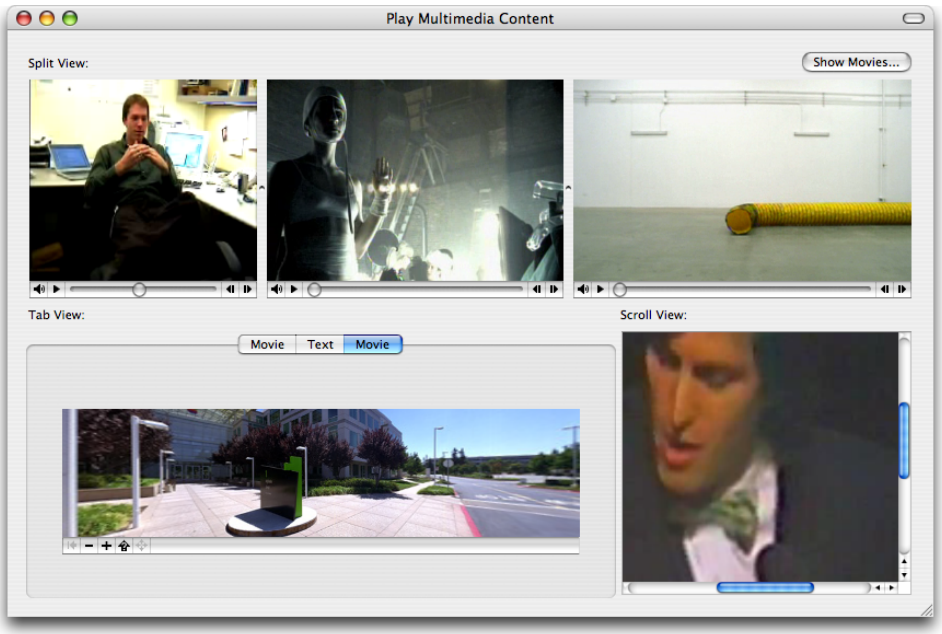
The Completed QTKit Multimedia Player

If you've worked through the steps outlined in this chapter, you'll have extended your knowledge of how to build a media player that is capable of displaying and playing back up to six QuickTime movies. You can modify the code to meet your own needs, if, for example, you want to set up the QTKitPlayer so that it only displays movies that are pre-selected. This is possible with a minimum amount of code tweaking.

The goal is to expand the possibilities that are available to you with the new QTKit framework.

The effect of multimedia playback can be dramatic, indeed, when the user is able to view all six movies at full screen on their computer and even resize each movie to meet their viewing needs. In the illustration shown in Figure 6-17, the QuickTime VR movie in the lower left portion of the playback window is stretched out for improved VR viewing and navigation.

Figure 6-17 Full screen multimedia playback with resizing of the QuickTime VR movie in the lower left portion of the window



Document Revision History

This table describes the changes to *QuickTime Kit Programming Guide*.

Date	Notes
2005-11-09	Added new task information in the sequence of steps required to build and extend the example QTKitPlayer application. Minor edits throughout, along with code formatting updates and fixes.
2005-04-29	Updated for public release of Mac OS X v10.4. Added chapters on how to implement advanced features in the QTKitPlayer sample application. First public version.

Document Revision History