QTKit Capture Programming Guide

QuickTime > Cocoa

2007-10-31

Contents

Introduction **Introduction to QTKit Capture Programming Guide 7**

Organization of This Document 7
See Also 8

Chapter 1 **Basics of Using QTKit Capture 9**

Tasks Supported by QTKit Capture Classes 9
How QTKit Capture Works 10
Using the QTKit Capture API 12
Base Classes 13
Input and Output Classes 13
Utility Classes 15
Device Access and User Interface Classes 15

Chapter 2 **Building a Simple QTKit Capture Application 17**

First Steps 17
- What You Need 18
- Prototype the Capture Player 18

Create the Project Using Xcode 3 19
- Name the Project Files and Import the QTKit Headers 21
- Determine the Actions and Outlets You Want 21

Create the User Interface Using Interface Builder 3 22
Set Up a Preview of the Captured Video Output 26
Wire the Start and Stop Buttons 27
Complete the Project Nib File in Xcode 32
Implement and Build Your Capture Application 34

Chapter 3 **Adding Audio Input and DV Camera Support 37**

Add Instance Variables 37
Modify Methods 37

Chapter 4 **Creating a QTKit Stop or Still Motion Application 41**

Set Up Your Project 41
Prototype the Still Motion Capture Application 42
Create the Project Using Xcode 3 42
- Import the QTKit Headers and Set Up Your Implementation File 44
- Determine the Actions and Outlets You Want 44

Create the User Interface Using Interface Builder 3 45

Prepare to Capture Single Frame Video 51
Complete the Project Nib File in Xcode 52
Implement and Build Your Still Motion Capture Application 56

Document Revision History 57

Figures

Chapter 1 **Basics of Using QTKit Capture 9**

Figure 1-1 Connecting inputs to outputs in a capture session 11
Figure 1-2 Handling multiple audio inputs in a capture session 12

Chapter 2 **Building a Simple QTKit Capture Application 17**

Figure 2-1 Prototype sketch of QTKit capture application 19
Figure 2-2 The Xcode 3 icon 19
Figure 2-3 The MyRecorder Xcode project window 20
Figure 2-4 The new Interface Builder 3 icon 22
Figure 2-5 QuickTime Capture View control in the library 22
Figure 2-6 The resized `QTCaptureView` object and its class Identity defined in the Identity Inspector 23
Figure 2-7 Setting the autosizing for your `QTCaptureView` object 23
Figure 2-8 Window attributes defined in the Inspector 24
Figure 2-9 Specifying Start and Stop push buttons 24
Figure 2-10 Setting up the autosizing for the Start and Stop buttons 25
Figure 2-11 The blue cube object for your controller 25
Figure 2-12 The object from the instantiated as a controller 26
Figure 2-13 The `mCaptureView` instance variable wired as an outlet for the `MyRecorderController` object 27
Figure 2-14 Wiring the recording buttons to the `MyRecorderController` object 28
Figure 2-15 Connecting the window to the `MyRecorderController` object as the delegate outlet 28
Figure 2-16 The window object wired up correctly to the delegate object 29
Figure 2-17 My Recorder controller object and buttons wired up correctly 29
Figure 2-18 Class actions and outlets specified in the identity inspector 30
Figure 2-19 The green light indicating synchronization between Xcode and Interface Builder 31
Figure 2-20 The completed user interface for the MyRecorder application 31
Figure 2-21 MyRecorder with Start and Stop buttons 35

Chapter 4 **Creating a QTKit Stop or Still Motion Application 41**

Figure 4-1 Prototype sketch of QTKit still motion capture application 42
Figure 4-2 The Xcode 3 icon 43
Figure 4-3 The StillMotion Xcode project window 43
Figure 4-4 The new Interface Builder 3 icon 45
Figure 4-5 The QuickTime Capture View object in the library 45
Figure 4-6 The QTCaptureView window and Class Identity in the Inspector 46
Figure 4-7 Setting the autosizing for your `QTCaptureView` object 46

Figure 4-8 The QTMovieView control in the Interface Builder library 47
Figure 4-9 The QTMovieView object defined in the field Class 47
Figure 4-10 The Window attributes in `MyDocument.nib` 48
Figure 4-11 Defining the size of the Window 49
Figure 4-12 Specifying the delegate outlet connection for File's Owner 49
Figure 4-13 Specifying the autosizing for the Add Frame button 50
Figure 4-14 Specifiying the actions and outlet connections in MyDocument.nib 50
Figure 4-15 The My Document Identity Inspector with File's Owner selected 51
Figure 4-16 Preparing to capture single video frames and outputting those frames to a QuickTime movie 52
Figure 4-17 Still motion recorded output as a three-frame QuickTime movie 56

Introduction to QTKit Capture Programming Guide

The QuickTime Kit is a Objective-C framework (`QTKit.framework`) with a rich API for manipulating time-based media. Introduced in Mac OS X v10.4, the QuickTime Kit provides a set of Objective-C classes and methods designed for the basic manipulation of media, including movie playback, editing, import and export to standard media formats, among other capabilities. With the release of Mac OS X v10.5 and the latest release of QuickTime 7, the reach and capability of the framework have been extended. The QuickTime Kit now includes the addition of 17 new classes, all designed to support professional-level video and audio capture, and pro-grade recording of media.

This guide describes the conceptual underpinnings of these new capture classes, and shows how to use the classes and methods in your application, through code examples and step-by-step tutorials.

If you are a QuickTime Cocoa developer who wants to integrate QuickTime movies in your application, you should read the material in this document. You don't necessarily need to be a seasoned Cocoa programmer to take advantage of the capabilities provided in this framework, although you'll need some prior experience working with Objective-C, Xcode, and Interface Builder to build and compile the code examples in the guide.

The various QuickTime and Cocoa mailing lists also provide a useful developer forum for raising issues and answering questions that are posted.

Organization of This Document

This document is organized into an overview chapter, followed by chapters that describe how you can build a simple QTKit capture application with an optimal number of lines of code, then extend that application with audio and DV support. Another chapter describes how you can construct an application that lets you capture single frames and produce stop or still motion video animation.

- "Basics of Using QTKit Capture" (page 9) introduces you to the key concepts associated with the QuickTime Kit capture architecture and describes how you can use the capture classes and methods in your application.
- "Building a Simple QTKit Capture Application" (page 17) discusses step-by-step how you can build a simple yet powerful QTKit capture player application that lets you capture a video stream and record the media to a QuickTime movie.
- "Adding Audio Input and DV Camera Support" (page 37) describes how you can extend the functionality of your QTKit capture player application by adding support for audio input and DV cameras with only a few lines of Objective-C code.
- "Creating a QTKit Stop or Still Motion Application" (page 41) describes how you can construct a simple still motion capture application that lets you capture a live video feed, grab frames one at a time with great accuracy, and then record the output of those frames to a QuickTime movie––with less than 100 lines of Objective-C code.

See Also

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

The following documents also provide helpful guides and references for many of the tasks described in this programming guide:

- *QuickTime Kit Framework Reference* contains the class and protocol reference documentation for the QTKit framework.
- *QuickTime Kit Programming Guide* shows how to build and extend a simple QTKitPlayer application, using Xcode 2.2 and Interface Builder.
- *Interface Builder User Guide* describes the latest version of Interface Builder 3.
- *Xcode Quick Tour for Mac OS X* provides an introduction on how to use the Xcode IDE.
- Cocoa Programming for Mac OS X, by Aaron Hillegass, is a useful guide for newcomers to the Objective-C programming language.

Note: This introductory and tutorial document is designed as companion text to the reference material in the *QuickTime Kit Framework Reference*. The various classes and methods in the QuickTime Kit framework are described in detail therein. It's a good idea, if possible, to have that document handy as you learn the API and work through the various steps you need to follow in building a QTKit capture application. However, you can still build the QTKit capture application described in "Building a Simple QTKit Capture Application" (page 17) without necessarily having to read through the reference.

Basics of Using QTKit Capture

The QuickTime Kit framework was developed by Apple to provide support for the most common media-related tasks of Cocoa and QuickTime 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 were new––but only where necessary. The goal was to provide high-level Cocoa interfaces for playing and editing, importing and exporting various types of media. In response to the growing needs of the developer community, new methods have been added with each release to the five base classes that comprise the framework.

Now with the introduction of Mac OS X v10.5 and the latest iteration of QuickTime 7, the QuickTime Kit framework has made a major leap forward, providing support for capturing media from external sources, such as cameras and microphones, and outputting that media to QuickTime movies. Fifteen new classes have been added to the existing five in the first iteration of the framework. The goal is to provide Cocoa and QuickTime developers with a viable and robust alternative to using the procedural C sequence grabber API, which allowed applications to obtain digitized data from external sources, such as video boards. Using the QTKit capture API is now the preferred way of developing applications that support capture and recording of media.

This chapter describes at a basic level the QTKit capture architecture and implementation available in Mac OS X v10.5. You'll gain an understanding of how you can capture, record, and output to various destinations by reading this chapter. Recording from an iSight camera or another DV device to a QuickTime file, for example, is one of the most common uses for the QTKit capture API.

To take advantage of this new API, you'll need to read this chapter first before moving ahead to the following chapters which describe how to build a QTKit capture player application.

Tasks Supported by QTKit Capture Classes

As a high-level Objective-C framework, the QuickTime Kit is built on top of a number of other Mac OS X graphics and imaging technologies, including QuickTime, Core Image, Core Audio, Core Animation, Quartz 2D, and OpenGL. This means much of the work involved in dealing with the processing of video, audio, and image media in your application is already provided for you by the underlying Mac OS X graphics and imaging engines, thereby reducing the code you need to write, as well as the code overhead required in your Xcode projects.

The new capture classes and methods available in the QuickTime Kit provide frame-accurate audio/video synchronization, and frame-accurate capture, meaning you can specify precisely––with timecodes––when you want capturing to occur. You also have access to transport controls of your camcorder, so you can fast forward and rewind the tape.

Using these classes and methods, you can capture media from one or more external sources, including

- Cameras
- Microphones

- Other external media devices, such as capture cards and tapedecks

The devices supported in Mac OS X v10.5 include:

- VDC over USB, including the built-in iSight camera
- IIDC over FireWire, includes external iSight
- HDV, with Final Cut Pro, devices (with the appropriate codecs installed)
- Core Audio HAL devices

Note that DV devices and Pro DV formats, such as DVCPro HD, require Final Cut Pro.

Important: With the introduction of the new, robust QTKit capture API, QuickTime developers are encouraged to move their development efforts away from usage of the component-based Sequence Grabber API.

After you've captured this media, you can record it to one or more output destinations, including but not necessarily limited to the following:

- A QuickTime movie (`.mov`) file
- A Cocoa view that previews video media captured from the input sources

Notably, as soon as you've captured media, you can also record the output to other destinations for use in custom-built applications and custom processing. This functionality is provided by the methods provided in the `QTCaptureDecompressedVideoOutput` and `QTCaptureVideoPreviewOutput` classes.

The next section discusses the types of capture objects you'll use in working with the QuickTime Kit framework. A basic understanding of these objects is important in building your QTKit capture application player.

How QTKit Capture Works

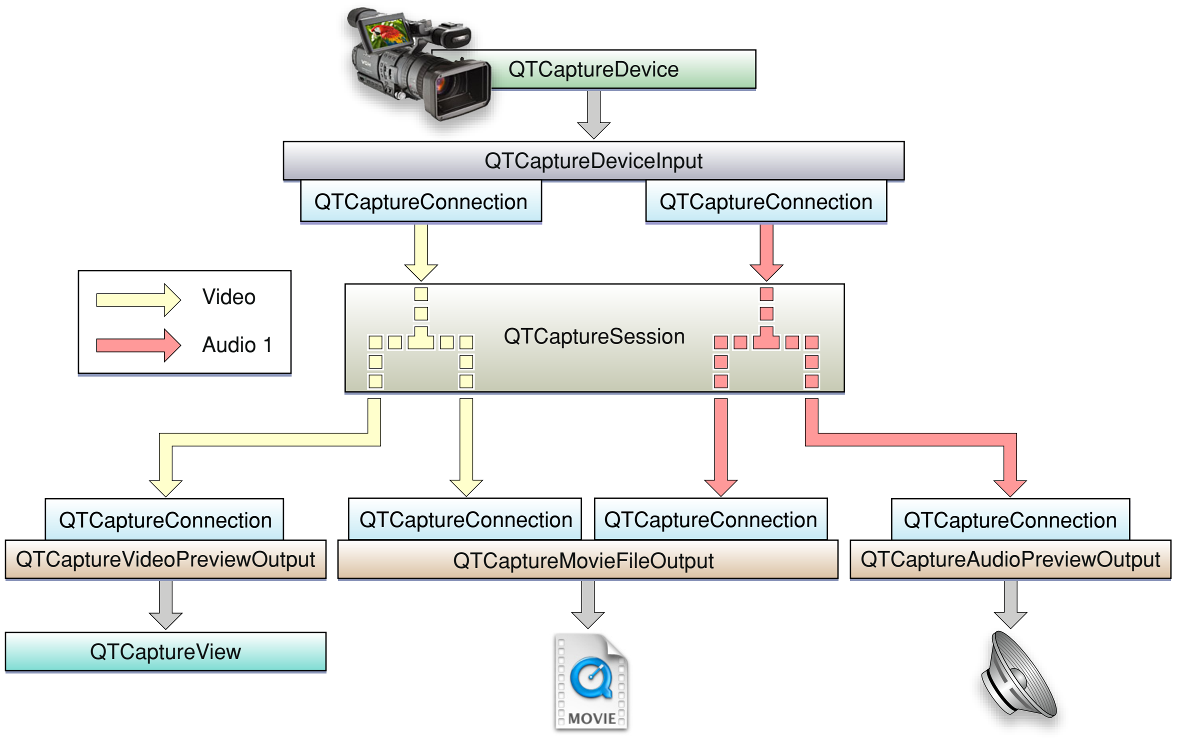
All QTKit capture applications make use of three basic types of objects: **capture inputs**, **capture outputs**, and a **capture session**. Capture inputs, which are subclasses of `QTCaptureInput`, provide the necessary interfaces to different sources of captured media.

A capture device input, which is a `QTCaptureDeviceInput` object––a subclass of `QTCaptureInput`––provides an interface to capturing from various audio/video hardware, such as cameras and microphones. Capture outputs, which are subclasses of `QTCaptureOutput`, provide the necessary interfaces to various destinations for media, such as QuickTime movie files, or video and audio previews.

A capture session, which is a `QTCaptureSession` object, manages how media that is captured from connected input sources is distributed to connected output destinations. Each input and output has one or more connection, which represents a media stream of a certain QuickTime media type, such as video or audio media. A capture session will attempt to connect all input connections to each of its outputs.

As shown in Figure 1-1 a capture session works by connecting inputs to outputs in order to record and preview video from a camera.

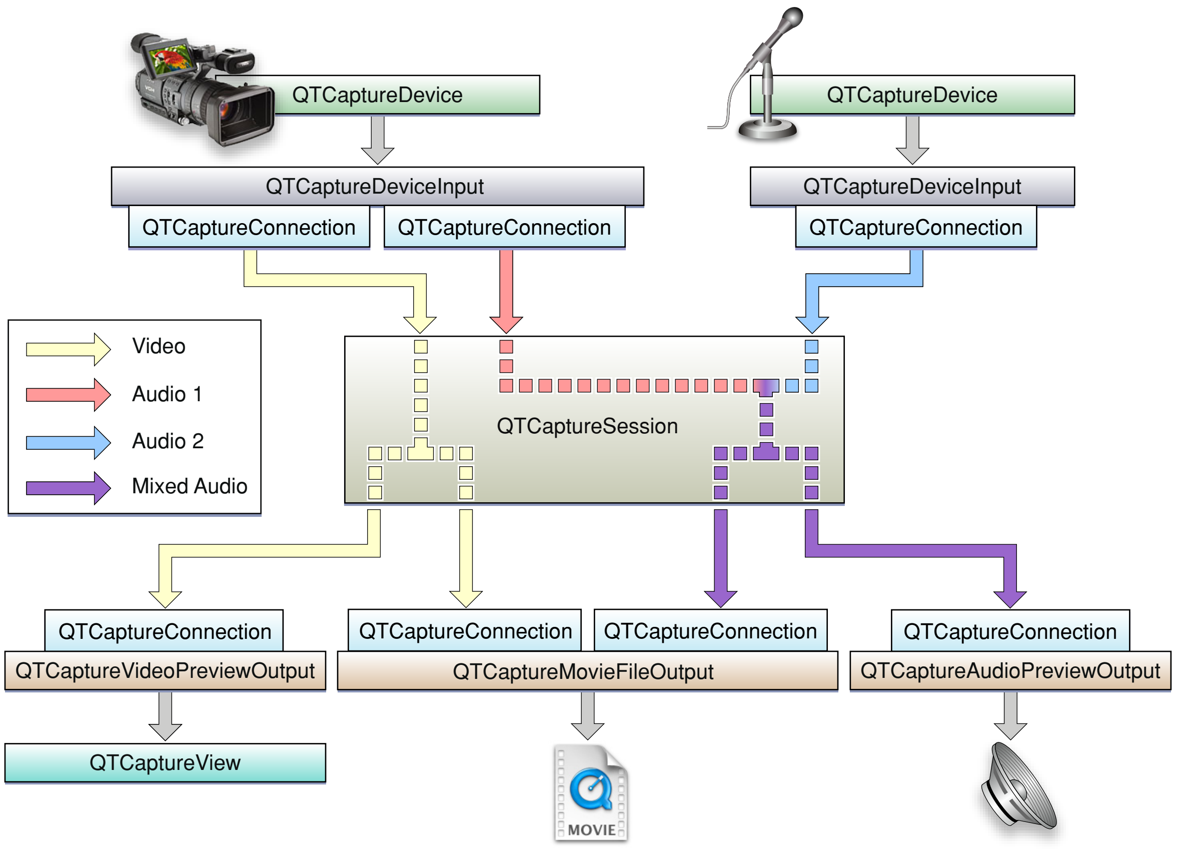
Figure 1-1 Connecting inputs to outputs in a capture session



A capture session works by distributing the video from its single video input connection to a connection owned by each output. In addition to distributing separate media streams to each output, the capture session is also responsible for mixing the audio from multiple inputs down to a single stream.

Figure 1-2 shows how the capture session handles multiple audio inputs.

Figure 1-2 Handling multiple audio inputs in a capture session



As illustrated in Figure 1-2, a capture session sends all of its input video to each output that accepts video and all of its input audio to each output that accepts audio. However, before sending the separate audio stream to its outputs, it mixes them down to one stream that can be sent to a single capture connection.

A capture session is also responsible for ensuring that all media are synchronized to a single time base in order to guarantee that all output video and audio are synchronized.

The connections belonging to each input and output are `QTCaptureConnection` objects. These describe the media type and format of each stream taken from an input or sent to an output. By referencing a specific connection, your application can have finer-grained control over which media enters and leaves a session. Thus, you can enable and disable specific connections, and control specific attributes of the media entering (for example, the volumes of specific audio channels).

Using the QTKit Capture API

The QTKit capture API is comprised of fifteen new classes and more than several hundred new methods, notifications, and attributes. To better understand how you can take advantage of this API in your Cocoa or QuickTime application, you may want to read this section describing the various groupings of the API. The complete description of all these classes and their associated methods is available in the *QuickTime Kit Framework Reference*.

There are four base classes, six classes devoted to input and output, another three utility classes, one class that deals with device input, another with the user interface, and finally, a class containing constants and error messages.

Base Classes

There are four classes in this group that can best be described as **base** classes. Understanding how these work is essential to using the QTKit capture API.

The `QTCaptureSession` class provides an interface for connecting input sources to output destinations. The method used most commonly in this class is `startRunning`, which tells the receiver to start capturing data from its inputs and then to send that data to its outputs. Notably, if you're using this method, when data does not need to be sent to file outputs, previews, or other outputs, your capture session should not be running, so that the overhead from capturing does not affect the performance of your application.

The `QTCaptureInput` and `QTCaptureOutput` classes, which are both abstract classes, provide interfaces for connecting inputs and outputs. An input source can have multiple connections, which is common for many cameras which have both audio and video output streams. Using `QTCaptureOutput` objects, you don't need to have a fixed number of connections, but you do need a destination for your capture session and all of its input data.

Class	Group	Tasks	Most commonly used methods
`QTCaptureSession`	Base	Primary interface for capturing media streams; manages connections between inputs and outputs; also manages when a capture is running.	`startRunning`, `addInput:error:`, `addOutput:error:`
`QTCaptureInput`	Base	Provides input source connections for a `QTCaptureSession`. Use subclasses of this class for inputs of a session.	`connections`
`QTCaptureOutput`	Base	Provides an interface for connecting capture output destinations, such as QuickTime files and video previews, to a `QTCaptureSession`.	`connections`
`QTCaptureConnection`	Base	Represents a connection over which a single stream of media data is sent from a `QTCaptureInput` to a `QTCaptureSession` and from a `QTCaptureSession` to a `QTCaptureOutput`.	`formatDescription`, `mediaType`, `setEnabled:`, `isEnabled`

Input and Output Classes

There are five **output** classes and only one **input** class belonging to this group.

You can use the methods available in the `QTCaptureDeviceInput` class to handle input sources for various media devices, such as cameras and microphones. The five output classes provide output destinations for `QTCaptureSession` objects that can be used to write captured media to QuickTime movies, for example, or to preview video or audio that is being captured. `QTCaptureFileOutput`, an abstract superclass, provides an output destination for a capture session to write captured media simply to files.

Class	Group	Tasks	Most commonly used methods
`QTCaptureAudio-PreviewOutput`	Input/Output	Represents an output destination for a`QTCaptureSession` that can be used to preview the audio being captured.	`volume`, `setVolume:`, `setOutputDevice-UniqueID:`
`QTCapture-DecompressedVideo-Output`	Input/Output	Represents an output destination for a `QTCaptureSession` object that can be used to process decompressed frames from the video being captured.	`setDelegate:`, `captureOutput: didOutputVideoFrame: withSampleBuffer: fromConnection:`
`QTCaptureDeviceInput`	Input/Output	Represents the input source for media devices, such as QuickTime files and video previews, to a `QTCaptureSession`.	`initWithDevice:`; returns an instance of `QTCaptureDeviceInput` associated with the given device.
`QTCaptureFileOutput`	Input/Output	Writes captured media to files and defines the interface for outputs that record media samples to files.	`recordToOutputFileURL:`, `setDelegate:`, `captureOutput: didFinishRecordingToOutputFileAtURL: forConnections: dueToError:`
`QTCaptureMovie-FileOutput`	Input/Output	Represents an output destination for a `QTCaptureSession` that writes captured media to QuickTime movie files.	`recordToOutputFileURL:`, `setDelegate:`, `captureOutput: didFinishRecordingToOutputFileAtURL: forConnections: dueToError:`
`QTCaptureVideo-PreviewOutput`	Input/Output	Represents an output destination for a`QTCaptureSession` that can be used to preview the video being captured.	`visualContextForConnection:`, `setDelegate:`, `captureOutput: didOutputVideoFrame: withSampleBuffer: fromConnection:`

Utility Classes

There are three classes belonging to this group: `QTCompressionOptions`, `QTFormatDescription`, and `QTSampleBuffer`. These are best characterized as **utility** classes, in that they perform tasks related to representing, for example, the compressions for particular media, or describing the formats of various media samples.

You can use `QTCompressionOptions` to describe compression options for all kinds of different media, using the `compressionOptionsIdentifiersForMediaType:` and `mediaType` methods. Compression options are created from presets keyed by a named identifier. These preset identifiers are listed in the *QuickTime Kit Framework Reference* in the chapter describing this class.

Using `QTSampleBuffer` objects, you can get information about sample buffer data that you may need to output or process the media samples in the buffer.

Class	Group	Tasks	Most commonly used methods
`QTCompressionOptions`	Utility	Represents a set of compression options for a particular type of media.	`compressionOptions-IdentifiersForMediaType:`, mediaType
`QTFormatDescription`	Utility	Describes the media format of media samples and of media sources, such as devices and capture connections.	`localizedFormatSummary`. The constant, `QTFormatDescription-VideoCleanAperture-DisplaySizeAttribute`
`QTSampleBuffer`	Utility	Provides format information, timing information, and metadata on media sample buffers.	`formatDescription`

Device Access and User Interface Classes

There are two classes in this particular group: `QTCaptureDevice` and `QTCaptureView`.

If you're working with `QTCaptureDevice` objects, your application can read any number of extended attributes available to this class, using the `deviceAttributes` and `attributeForKey:` methods. Beyond that, you can use key-value coding to get and set attributes. If you wish to observe changes for a given attribute, you can add a key-value observer where the key path is the attribute key. Note that you cannot create instances of `QTCaptureDevice` directly.

You can use the methods available in the `QTCaptureView` class, which is a subclass of `NSView`, to preview video that is being processed by an instance of `QTCaptureSession`. The class creates and maintains its own `QTCaptureVideoPreviewOutput` to gather the preview video you need from the capture session.

Class	Group	Tasks	Most commonly used methods
`QTCaptureDevice`	Device Access and UI	Represents an available capture device.	`inputDevices`, `open:`, `isOpen`, `close`, `localizedDisplayName`
`QTCaptureView`	Device Access and UI	Displays a video preview of a capture session.	`setCaptureSession:`

Building a Simple QTKit Capture Application

In this chapter, you'll build a QTKit capture player, a simple yet powerful application that demonstrates how you can take advantage of some of the new capture classes and methods available in the next iteration of the QuickTime Kit framework. When completed, your QTKit capture player application will allow you to capture a video stream and record the media to a QuickTime movie. You won't have to write more than 20 or 30 lines of Objective-C code to implement this capture player.

Using Xcode 3 as your integrated development environment (IDE), along with Interface Builder 3, you'll see how easy it is to work with the QuickTime Kit framework. In this example, you'll use the new QTKit capture control provided in the library of controls available in Interface Builder 3. The QTKit capture control will perform much of the work for you in implementing the design of the user interface for this application.

Following the steps in this guide, you'll be able to build a functioning capture player application that controls the capture of QuickTime movies, adding simple start and stop buttons, and allowing you to output and display your captured files in QuickTime Player. For this project, you'll need an iSight camera, either built-in or plugged into your Macintosh. You'll also need Mac OS X v10.5, the latest release of Mac OS X, installed in your system.

In building your QTKit capture application, you'll work with the following three classes:

- `QTCaptureSession`. The primary interface for capturing media streams.
- `QTCaptureMovieFileOutput`. The output destination for a QTCaptureSession object that writes captured media to QuickTime movie files.
- `QTCaptureDeviceInput`. The input source for media devices, such as cameras and microphones.

For purposes of this tutorial, you won't need to have a complete understanding of the methods that belong to these capture classes. As you extend your knowledge of the QTKit framework, you should refer to the *QuickTime Kit Framework Reference*, which describes in detail the methods, notifications, attributes, constants, and types that comprise the collection of classes in the QTKit API.

First Steps

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 QTKit capture player application should serve as 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 capture player application.

What You Need

Before you get started with your QTKit capture player project, be sure that you are running Mac OS X v10.5 and have the following items installed on your system:

- Xcode 3 and Interface Builder 3. Note that although you can use Xcode 2.2 to build your project, to take full advantage of the new programming features available, you'll want to use Xcode 3. Also, Interface Builder 3 provides a new paradigm for working with and building the user interface for your application. Palettes are no longer provided; instead, you'll work with a new library of controls that are designed to enable you to hook up the components of your user interface with greater ease and efficiency. In the end, you'll be able to build applications faster and take advantage of this rich, new library of controls.
- The QuickTime Kit framework, which resides in the Mac OS X v10.5 `System/Library/Frameworks` directory as `QTKit.framework`.
- An iSight camera connected to your computer.

Important: The complete sample code discussed in this and the next chapter is available for download at Sample Code: QuickTime Cocoa, as `MyRecorder`.

Note: The `MyRecorder` sample code, as explained in this chapter, will not support DV cameras, which are of type `QTMediaTypeMuxed`, rather than `QTMediaTypeVideo`. The next chapter in this programming guide, "Adding Audio Input and DV Camera Support" (page 37), explains how you can add code that lets you work with DV cameras and use the `QTMediaTypeMuxed` type.

Prototype the Capture Player

Interface Builder lets you specify the windows, menus, and views of your application, while Xcode enables you to define the behavior behind them. Interface Builder provides the basic support you need for configuring the items in your user interface. Beyond that, most of the work you do in constructing your application takes place in Xcode.

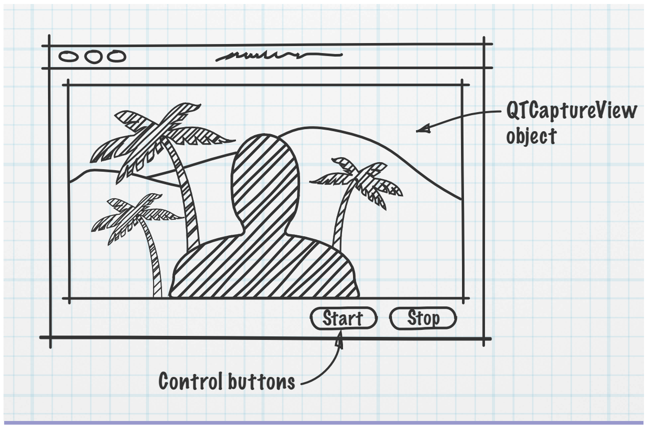
When designing your application, start by defining your application's data model in Xcode. Once you've constructed a workable data model, you can use Interface Builder to create a set of basic windows, menus, and views for presenting that data. Depending on the complexity of your design, you may also need to create custom views and controls, and then integrate them into Interface Builder and add them to your nib files.

Creating the controller objects and tying them to your data model in your user interface is the final step in the design process.

Of course, you can just jump right in and start assembling windows and menus in Interface Builder. However, using Interface Builder 3, which is the latest iteration, it's important to have a good understanding of your application's desired behavior first. Knowing your application's data model, and knowing what operations will occur on that data, will help you piece together the design elements you need to show in order to convey that information to the end user.

You may want to start by creating a rough sketch of your QTKit capture application. Think of what design elements you want to incorporate into the application. Rather than simply jumping into Interface Builder and doing your prototype there, you may want to visualize the elements first in your rough sketch, as shown in Figure 2-1.

Figure 2-1 Prototype sketch of QTKit capture application



In this design prototype, you can start with three simple objects: a capture view and two control buttons. These will be the building blocks for your application. After you've sketched them out, you can begin to think of how you'll be able to hook them up in Interface Builder and what code you need in your Xcode project to make this happen.

Create the Project Using Xcode 3

To create the project, follow these steps:

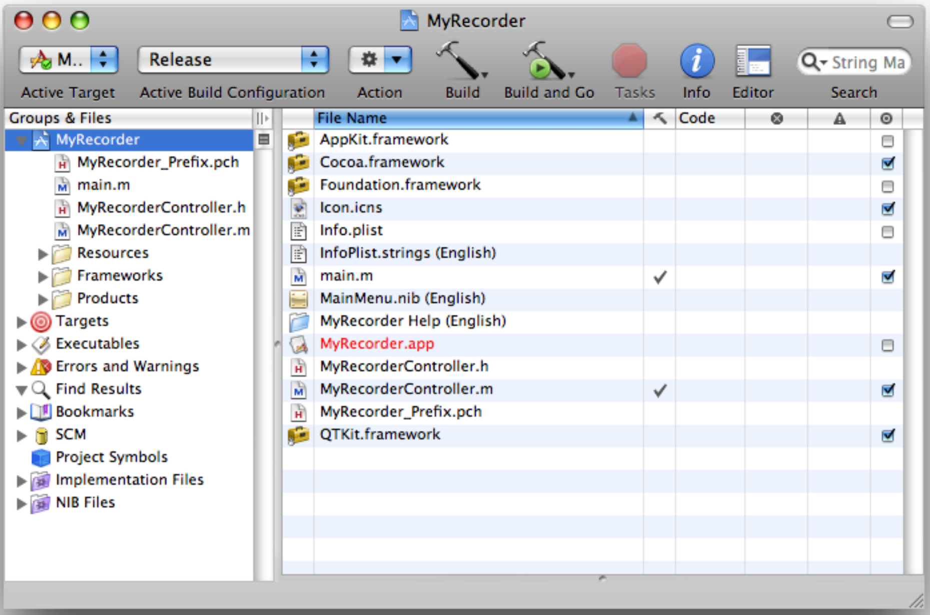
1. Launch Xcode 3 (shown in Figure 2-2) and choose File > New Project.

   Figure 2-2 The Xcode 3 icon

2. When the new project window appears, select Cocoa Application.

3. Name the project `MyRecorder` and navigate to the location where you want the Xcode application to create the project folder. Now the Xcode project window appears, as shown in Figure 2-3.

Figure 2-3 The MyRecorder Xcode project window



Groups & Files
- MyRecorder
  - MyRecorder_Prefix.pch
  - main.m
  - MyRecorderController.h
  - MyRecorderController.m
  - Resources
  - Frameworks
  - Products
- Targets
- Executables
- Errors and Warnings
- Find Results
- Bookmarks
- SCM
- Project Symbols
- Implementation Files
- NIB Files

File Name		Code			
AppKit.framework					☐
Cocoa.framework					☑
Foundation.framework					☐
Icon.icns					☑
Info.plist					☐
InfoPlist.strings (English)					
main.m	✓				☑
MainMenu.nib (English)					
MyRecorder Help (English)					
MyRecorder.app					☐
MyRecorderController.h					
MyRecorderController.m	✓				☑
MyRecorder_Prefix.pch					
QTKit.framework					☑

4. Next, you need to add the QuickTime Kit framework to your `MyRecorder` project. Although obvious, this step is sometimes easy to forget. Note that you don't need to add the QuickTime framework to your project, just the QuickTime Kit framework. Choose Project > Add to Project.

5. The QuickTime Kit framework resides in the `System/Library/Frameworks` directory. Select `QTKit.framework`, and click Add when the Add To Targets window appears to add it to your project.

Important: This completes the first sequence of steps in your project. In the next sequence, you'll move ahead to define actions and outlets in Xcode *before* working with Interface Builder. This may involve something of a paradigm shift in how you may be used to building and constructing an application with versions of Interface Builder prior to Interface Builder 3. Because you've already prototyped your QTKit capture application, at least in rough form with a clearly defined data model, you can now determine which actions and outlets need to be implemented. In this case, you have a `QTCaptureView` object, which is a subclass of `NSView`, and two simple buttons to start and stop the recording of your captured media content.

Name the Project Files and Import the QTKit Headers

1. Choose File > New File. In the panel, scroll down and select Cocoa > Objective-C class, which includes the `Cocoa.Cocoa.h`. files.
2. Name your implementation file `MyRecorderController.m`. You'll also check the item to name your declaration file `MyRecorderController.h`.
3. In your `MyRecorderController.h` file, add `#import <QTKit/QTkit.h>`.

Determine the Actions and Outlets You Want

1. Now you can begin adding outlets and actions. In your `MyRecorderController.h` file, add the instance variable `mCaptureView` in the following line of code:

   ```
   IBOutlet QTCaptureView *mCaptureView;
   ```

2. You also want to add these two actions:

   ```
   - (IBAction)startRecording:(id)sender;
   - (IBAction)stopRecording:(id)sender;
   ```

3. Now open your `MyRecorderController.m` file and add the following actions, along with the requisite braces:

```
- (IBAction)startRecording:(id)sender
{
}
- (IBAction)stopRecording:(id)sender
{
}
```

At this point the code in your `MyRecorderController.h` file should look like this.

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

@interface MyRecorderController : NSObject {
    IBOutlet QTCaptureView *mCaptureView;
}
- (IBAction)startRecording:(id)sender;
- (IBAction)stopRecording:(id)sender;

@end
```

This completes the second stage of your project. Now you'll need to shift gears and work with Interface Builder 3 to construct the user interface for your project.

Create the User Interface Using Interface Builder 3

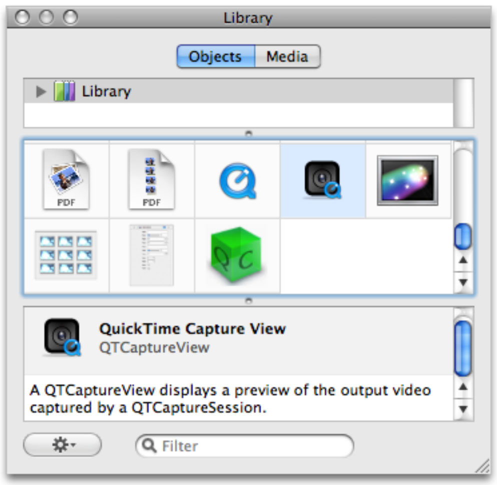
In the next phase of your project you'll see how seamlessly Interface Builder and Xcode work together, enabling you to construct and implement the various elements in your project more efficiently and with less overhead.

1. Open Interface Builder 3 (Figure 2-4) and drag the `MainMenu.nib` file in your Xcode project window on the Interface Builder 3 icon. Because of the new integration between Xcode 3 and Interface Builder 3, you'll find the actions and outlets you've declared in your `MyRecorderController.h` file are also synchronously updated in Interface Builder 3. This will become apparent once you open your nib file and begin to work with the library of controls available in Interface Builder 3.

   Figure 2-4 The new Interface Builder 3 icon

2. In Interface Builder 3, you'll find a new library of controls. Scroll down until you find the QuickTime Capture View control, as shown in Figure 2-5.

   Figure 2-5 QuickTime Capture View control in the library

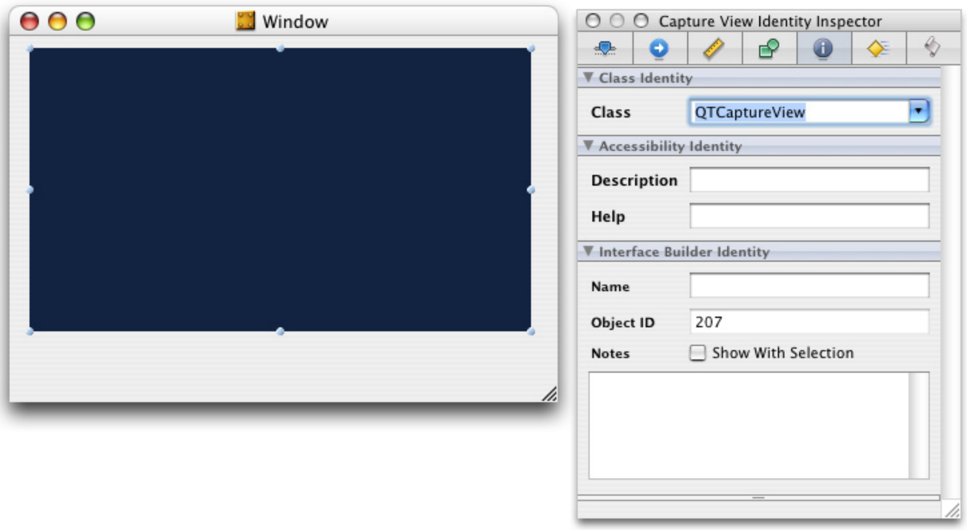
   A QTCaptureView displays a preview of the output video captured by a QTCaptureSession.

   The `QTCaptureView` object provides you with an instance of a view subclass to display a preview of the video output that is captured by a capture session.

3. Drag the `QTCaptureView` object into your window and resize the object to fit the window, allowing room for the two Start and Stop buttons in your QTKit capture player.

4. Choose Tools > Inspector. In the Identity Inspector, select the information ("i") icon. Click in the field Class and your `QTCaptureView` object appears, as shown in Figure 2-6.

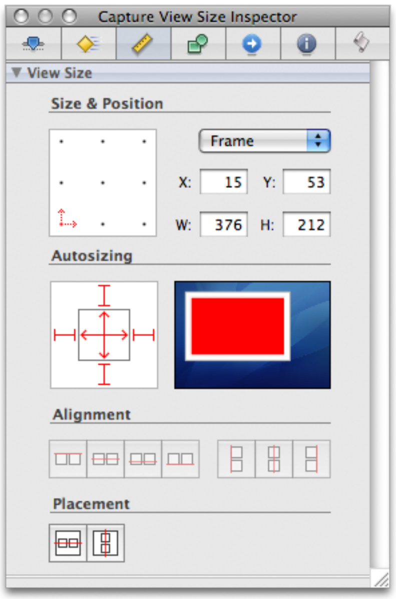
Figure 2-6 The resized `QTCaptureView` object and its class Identity defined in the Identity Inspector



Class: QTCaptureView
Object ID: 207

5. Set the autosizing for the object in the Capture View Size Inspector, as shown in Figure 2-7.

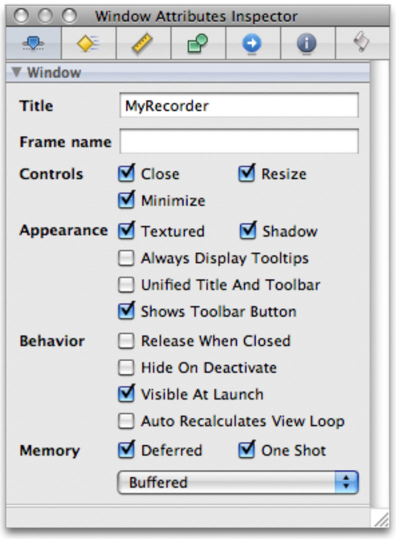
Figure 2-7 Setting the autosizing for your `QTCaptureView` object



Frame
X: 15 Y: 53
W: 376 H: 212

6. Define the attributes of your MyRecorder Window, as shown in Figure 2-8.

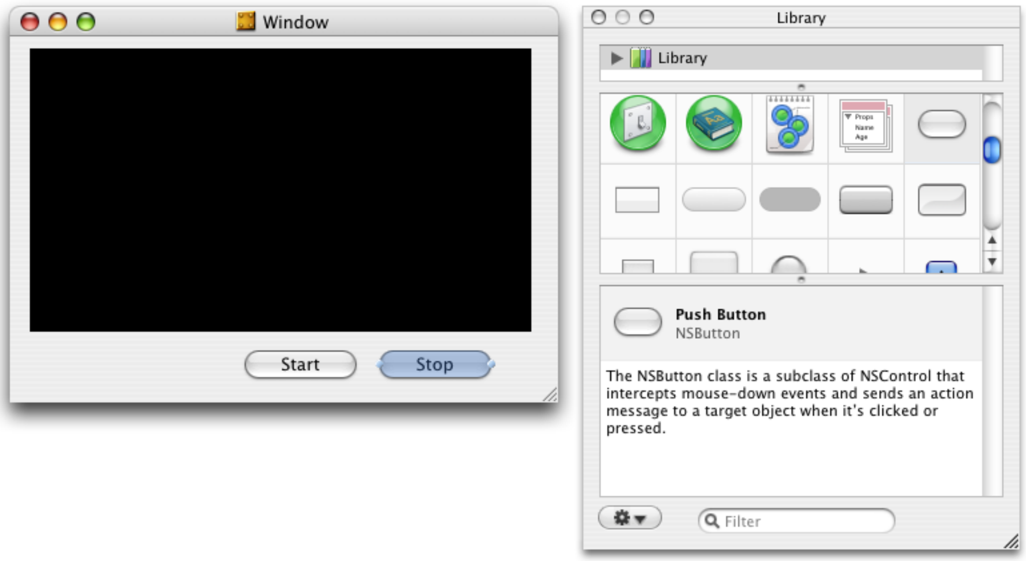
Figure 2-8 Window attributes defined in the Inspector



Title: MyRecorder
Frame name:
Controls: [✓ Close] [✓ Resize] [✓ Minimize]
Appearance: [✓ Textured] [✓ Shadow] Always Display Tooltips; Unified Title And Toolbar; [✓ Shows Toolbar Button]
Behavior: Release When Closed; Hide On Deactivate; [✓ Visible At Launch]; Auto Recalculates View Loop
Memory: [✓ Deferred] [✓ One Shot]; Buffered

7. In the Library, select the Push Button control and drag it to the Window, as shown in Figure 2-9. Enter the text **Start** and duplicate the button to create another button as **Stop**. In autosizing, set the struts for both buttons at the bottom and right outside corners, leaving the inside struts untouched.

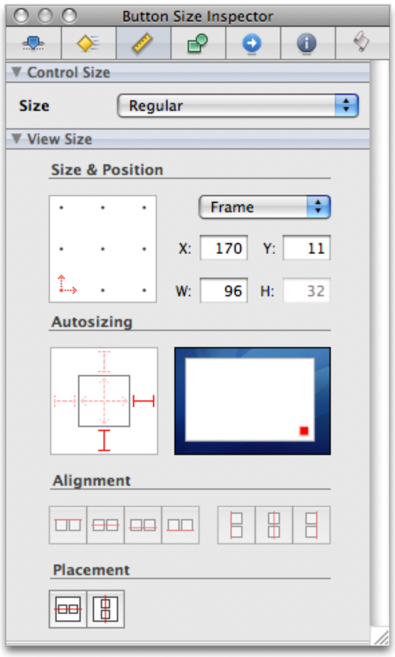
Figure 2-9 Specifying Start and Stop push buttons



The NSButton class is a subclass of NSControl that intercepts mouse-down events and sends an action message to a target object when it's clicked or pressed.

8. Set up the autosizing for your buttons by selecting the button and clicking the Button Size Inspector shown in Figure 2-10.

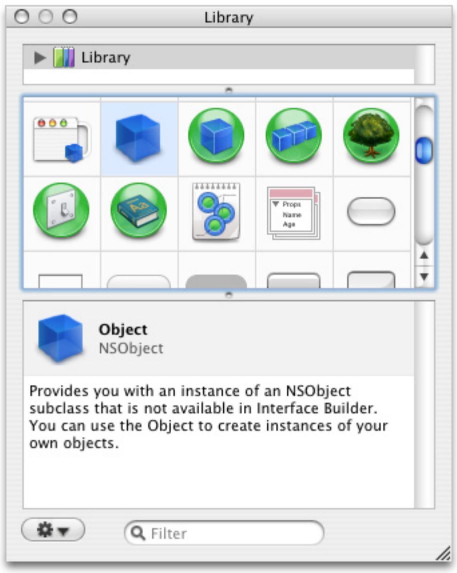
Figure 2-10 Setting up the autosizing for the Start and Stop buttons



Size: Regular
Frame
X: 170 Y: 11
W: 96 H: 32

9. In the Library, scroll down and select the blue cube control shown in Figure 2-11, which is an object (`NSObject`) you can instantiate as your controller.

Figure 2-11 The blue cube object for your controller

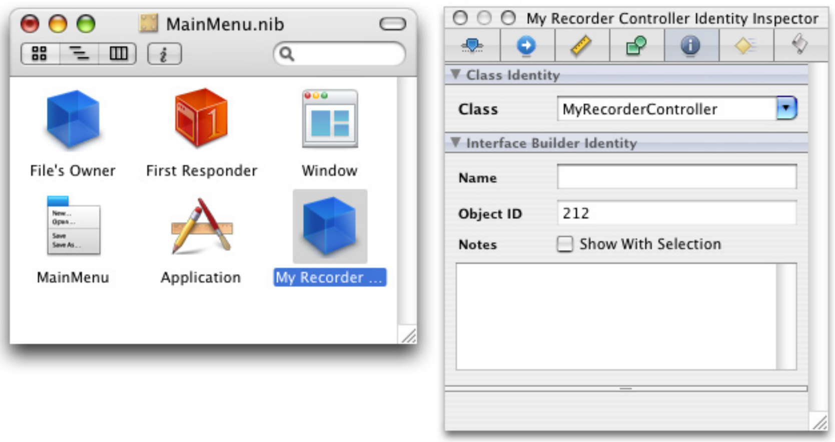


Object
NSObject

Provides you with an instance of an NSObject subclass that is not available in Interface Builder. You can use the Object to create instances of your own objects.

10. Drag the object into your `MainMenu.nib` window, as shown in Figure 2-12.

Figure 2-12 The object from the instantiated as a controller



Class: MyRecorderController
Name:
Object ID: 212
Notes:

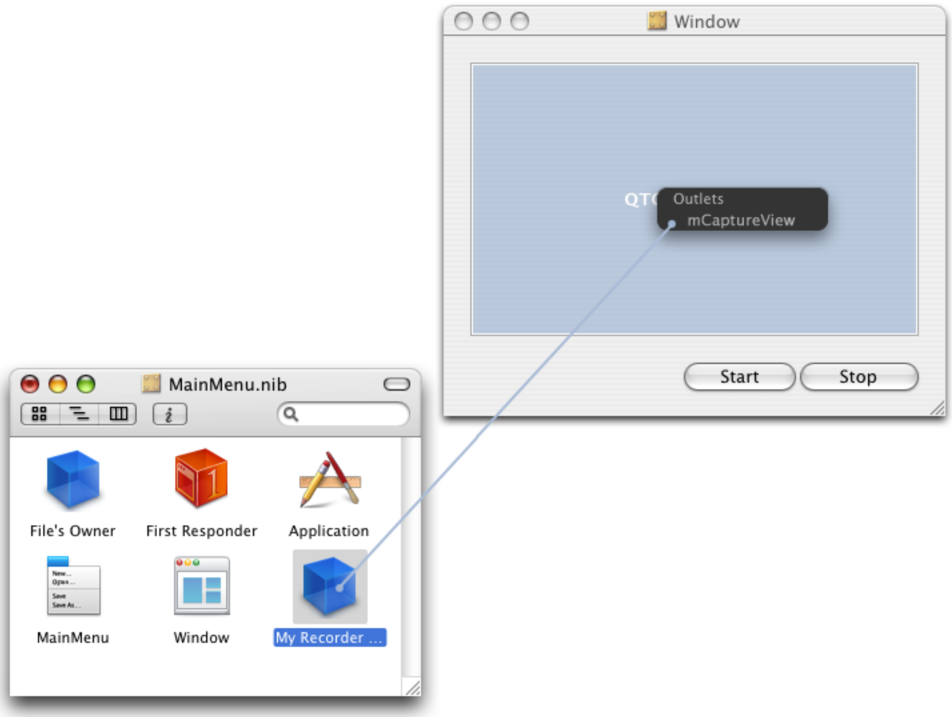
11. Select the object and enter its name as **My Recorder Controller**. Then click the information icon in the Inspector. When you click the Class Identity field, the `MyRecorderController` object appears. Interface Builder has automatically updated the `MyRecorderController` class specified in your Xcode implementation file. You don't need to enter the name of this class in the Class Identity field. Note that to verify and reconfirm that an update has occurred, press Return. If the identify field is not automatically updated, you may need to specify manually that it is a `MyRecorderController` object.

Set Up a Preview of the Captured Video Output

In the next phase of your project, you'll see how seamlessly Interface Builder and Xcode work together, enabling you to construct and implement the various elements in your project more efficiently and with less overhead.

1. In Interface Builder, hook up the `MyRecorderController` object to the `QTCaptureView` object. Control-drag from the `MyRecorderController` object in your nib file to the `QTCaptureView` object. A transparent panel will appear, as shown in Figure 2-13, displaying the `IBOutlet` instance variable, `mCaptureView`, that you've specified in your declaration file.

Figure 2-13 The `mCaptureView` instance variable wired as an outlet for the `MyRecorderController` object



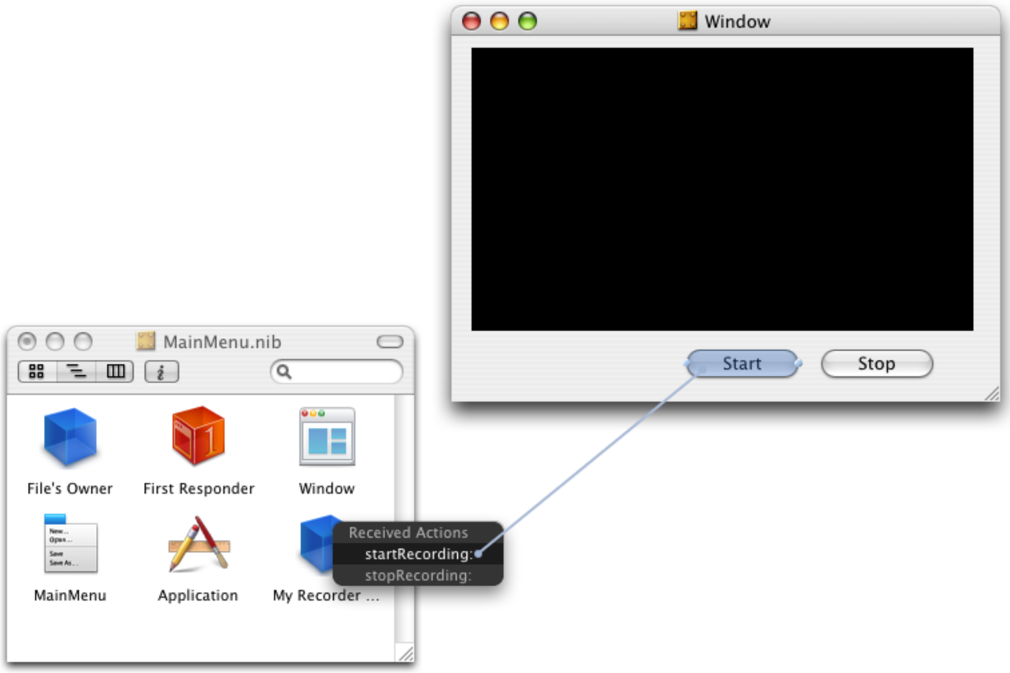
2. Click the Interface Builder outlet `mCaptureView` to wire up the two objects.

Wire the Start and Stop Buttons

Now you're ready to add your **Start** and **Stop** push buttons and wire them up in your MainMenu.nib window.

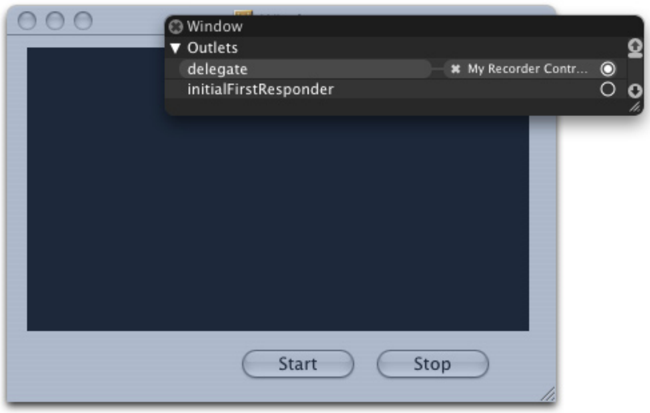
1. Control-drag each of the **Start** and **Stop** buttons from the window to the MyRecorderController object, as shown in Figure 2-14. Click the `startRecording:` method in the transparent Received Actions panel to connect the Start button, and likewise, the `stopRecording:` method in the Received Actions panel to connect the Stop button.

Figure 2-14 Wiring the recording buttons to the `MyRecorderController` object



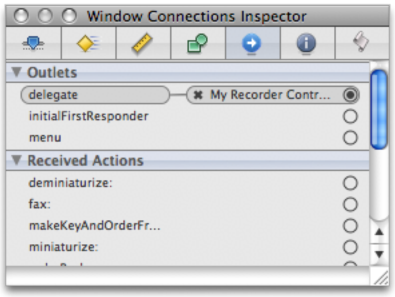
2. Now you'll need to hook up the window and the `MyRecorderController` object as a delegate, shown in Figure 2-15. Control-drag a connection from the window to the `MyRecorderController` object and click the outlet, connecting the two objects.

Figure 2-15 Connecting the window to the `MyRecorderController` object as the delegate outlet



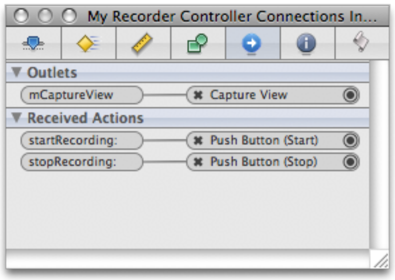
3. To verify that you've correctly wired up your window object to your delegate object, select the Window and click the Window Connections Inspector icon, shown in Figure 2-16.

Figure 2-16 The window object wired up correctly to the delegate object



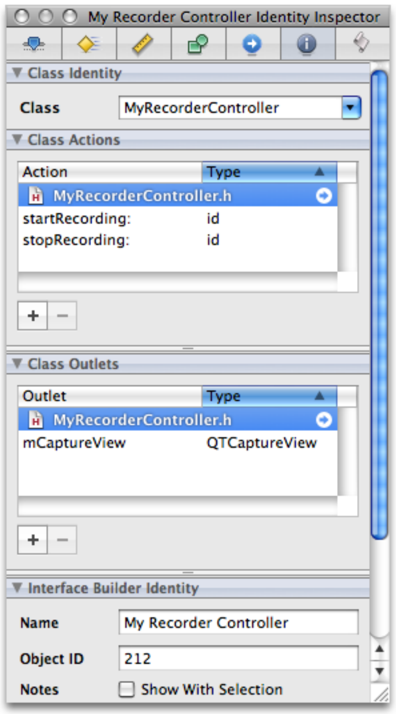
4. Verify that you've correctly wired up your outlets and received actions. Select the My Recorder Controller object and click the My Recorder Controller Connections Inspector icon, shown in Figure 2-17.

Figure 2-17 My Recorder controller object and buttons wired up correctly



5. Check the My Recorder Controller Identity Inspector panel to confirm the class actions and class outlets, shown in Figure 2-18.

Figure 2-18 Class actions and outlets specified in the identity inspector


My Recorder Controller Identity Inspector
Class Identity
Class Actions
Class Outlets
Interface Builder Identity
Show With Selection

Class: MyRecorderController

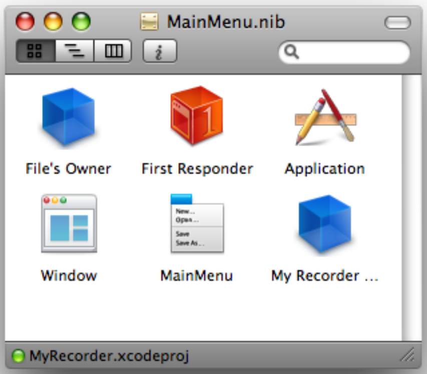
Action	Type
MyRecorderController.h	
startRecording:	id
stopRecording:	id

Outlet	Type
MyRecorderController.h	
mCaptureView	QTCaptureView

Name: My Recorder Controller
Object ID: 212
Notes: Show With Selection

6. Click the `MainMenu.nib` file to verify that Interface Builder and Xcode have worked together to synchronize the actions and outlets you've specified. A small green light appears at the left bottom corner of the `MainMenu.nib` file next to `MyRecorder.xcodeproj` to confirm this synchronization, as shown in Figure 2-19.

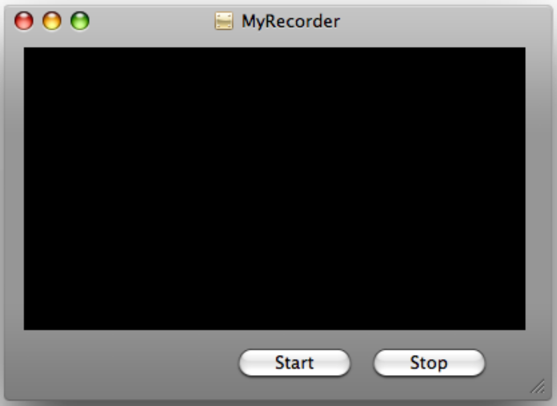
Figure 2-19 The green light indicating synchronization between Xcode and Interface Builder



7. Save your nib file.

8. Verify that your QTKit MyRecorder capture application appears as shown in Figure 2-20.

Figure 2-20 The completed user interface for the MyRecorder application



You've now completed your work in Interface Builder 3. In this next sequence of steps, you'll return to your Xcode project, adding a few lines of code in both your declaration and implemention files to build and compile the QTKit capture player application.

Complete the Project Nib File in Xcode

To complete the project nib file, you'll need to declare the outlet you set up and connected in Interface Builder, and define the instance variables that point to the capture session, as well as to the input and output objects.

1. In your Xcode project, you need to add the instance variables to the interface declaration. Add these lines of code in your `MyRecorderController.h` declaration file:

```
@interface MyRecorderController : NSObject {
QTCaptureSession            *mCaptureSession;
QTCaptureMovieFileOutput    *mCaptureMovieFileOutput;
QTCaptureDeviceInput        *mCaptureDeviceInput;
```

   The `mCaptureSession` instance variable points to the `QTCaptureSession` object, and the `mCaptureMovieFileOutput` instance variable points to the `QTCaptureMovieFileOutput` object. The last line declares that the `mCaptureDeviceInput` instance variable points to the `QTCaptureDeviceInput` object.

   The complete code for your declaration file should look like this:

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

@interface MyRecorderController : NSObject {

    IBOutlet QTCaptureView *mCaptureView;

    QTCaptureSession            *mCaptureSession;
    QTCaptureMovieFileOutput    *mCaptureMovieFileOutput;
    QTCaptureDeviceInput        *mCaptureDeviceInput;


}
- (IBAction)startRecording:(id)sender;
- (IBAction)stopRecording:(id)sender;

@end
```

2. In your `MyRecorderController.m` implementation file, add these lines of code, following your `@implementation MyRecordController` directive:

   Important: There is a specific, though not rigid, order of steps you want to follow in constructing your code. These are the steps you need to follow:

   a. Create the capture session.

b. Find the device and create the device input. Then add it to the session.

c. Create the movie file output and add it to the session.

d. Associate the capture view in the user interface with the session.

```
- (void)awakeFromNib
{
//Create the capture session
    mCaptureSession = [[QTCaptureSession alloc] init];

//Connect inputs and outputs to the session
    BOOL success = NO;
    NSError *error;

// Find a video device
QTCaptureDevice *device = [QTCaptureDevice
defaultInputDeviceWithMediaType:QTMediaTypeVideo];
    if (device) {
        success = [device open:&error];
        if (!success) {
            // Handle error
        }
// Add the video device to the session as device input
        mCaptureDeviceInput = [[QTCaptureDeviceInput alloc] initWithDevice:device];
        success = [mCaptureSession addInput:mCaptureDeviceInput error:&error];
        if (!success) {
            // Handle error
        }
// Create the movie file output and add it to the session
    mCaptureMovieFileOutput = [[QTCaptureMovieFileOutput alloc] init];
    success = [mCaptureSession addOutput:mCaptureMovieFileOutput error:&error];
    if (!success) {
        // Handle error
    }
// Set the controller be the movie file output delegate.
    [mCaptureMovieFileOutput setDelegate:self];

// Associate the capture view in the UI with the session

    [mCaptureView setCaptureSession:mCaptureSession];
    }
// Start the capture session running
        [mCaptureSession startRunning];

}
```

3. Add these lines to handle window closing notifications for your device input and stop the capture session.

```
- (void)windowWillClose:(NSNotification *)notification
{
    [mCaptureSession stopRunning];
    [[mCaptureDeviceInput device] close];

}
```

4. Insert the following block of code to handle deallocation of memory for your capture objects.

```
 - (void)dealloc
{
    [mCaptureSession release];
    [mCaptureDeviceInput release];
    [mCaptureMovieFileOutput release];

    [super dealloc];
}
```

5. Implement these start and stop actions, then add the following lines of code to specify the output destination for your recorded media, in this case a QuickTime movie (`.mov`) in your `/Users/Shared` folder.

```
- (IBAction)startRecording:(id)sender
{
    [mCaptureMovieFileOutput recordToOutputFileURL:[NSURL
fileURLWithPath:@"/Users/Shared/My Recorded Movie.mov"]];
}

- (IBAction)stopRecording:(id)sender
{
    [mCaptureMovieFileOutput recordToOutputFileURL:nil];
}
```

6. Add these lines of code to finish recording and then launch your recording as a QuickTime movie on your Desktop.

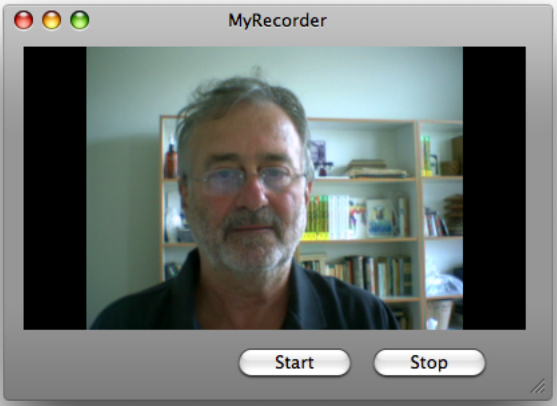
```
- (void)captureOutput:(QTCaptureFileOutput *)captureOutput
didFinishRecordingToOutputFileAtURL:(NSURL *)outputFileURL forConnections:(NSArray
 *)connections dueToError:(NSError *)error
{
    [[NSWorkspace sharedWorkspace] openURL:outputFileURL];
    // Do something with the movie at /Users/Shared/My Recorded Movie.mov
}
```

Implement and Build Your Capture Application

After you've saved your project, click Build and Go. After compiling, click the **Start** button to record, and the **Stop** button to stop recording. The output of your captured session is saved as a QuickTime movie in the path you've specified in this code sample.

Now you can begin capturing and recording with your QTKit capture player application, as shown in Figure 2-21. Using a simple iSight camera, you can capture and record media, and then output your recording to a QuickTime movie. .

Figure 2-21 MyRecorder with Start and Stop buttons



In the next chapter you'll see how easy it is to add audio input capability to your QTKit capture player application. Only a half dozen lines of code are required. You can build on what you've already written for your `MyRecorder` project, and add audio, along with support for DV cameras other than your iSight camera, with a minimum of programming effort.

Adding Audio Input and DV Camera Support

If you've worked through the sequence of steps outlined in the previous chapter, you're now ready to extend the functionality of your QTKit capture player application.

In this chapter, you'll add audio input capability to your capture application, as well as support for input from DV cameras other than your built-in or attached iSight camera. This is accomplished with only a dozen lines of Objective-C code, with error handling included.

Add Instance Variables

Follow these steps to add audio input capability and video input from DV cameras.

1. Launch Xcode 3 and open your `MyRecorder` project. Click the `MyRecorderController.h` declaration file. You need to modify and add the following instance variables, so the code looks like this:

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

@interface MyRecorderController : NSObject {
    IBOutlet QTCaptureView      *mCaptureView;

    QTCaptureSession            *mCaptureSession;
    QTCaptureMovieFileOutput    *mCaptureMovieFileOutput;
    QTCaptureDeviceInput        *mCaptureVideoDeviceInput;
    QTCaptureDeviceInput        *mCaptureAudioDeviceInput;
}
- (IBAction)startRecording:(id)sender;
- (IBAction)stopRecording:(id)sender;

@end
```

2. Notably, you've added two instance variables that point to the `QTCaptureDeviceInput` class. These are the audio and video input device variables that enable you to capture audio, as well as video from external DV cameras.

```
    QTCaptureDeviceInput             *mCaptureVideoInputDevice;
    QTCaptureDeviceInput             *mCaptureAudioInputDevice;
```

Modify Methods

Now open your `MyRecorderController.m` implementation file.

1. Scroll down to the code block that begins with `// Find a video device`. After the following block of code, which you need in order to find a video device, such as the iSight camera, you'll add a new block.

```
// Find a video device
QTCaptureDevice *videoDevice = [QTCaptureDevice
defaultInputDeviceWithMediaType:QTMediaTypeVideo];
success = [videoDevice open:&error];
```

2. Add this block, which enables you to find and open a muxed video input device, such as a DV camera. (Note that in a muxed video, the audio and video tracks are mixed together.)

```
// If a video input device can't be found or opened, try to find and open a
muxed input device
    if (!success) {
        videoDevice = [QTCaptureDevice
defaultInputDeviceWithMediaType:QTMediaTypeMuxed];
        success = [videoDevice open:&error];
}
if (!success) {
   [videoDevice = nil;
    // Handle error
}
if (videoDevice) {
```

3. Scroll down to the block of code that begins with the comment `//Add the video device to the session as a device input`. After that block, add the following lines, which add support for audio from an audio input device. Note that you've added an audio type of `QTMediaTypeSound` to the video device to handle the chores of capturing your audio stream in your capture session.

```
        mCaptureVideoDeviceInput = [[QTCaptureDeviceInput alloc]
initWithDevice:videoDevice];
        success = [mCaptureSession addInput:mCaptureVideoDeviceInput
error:&error];
        if (!success) {
            // Handle error
        }

    // If the video device doesn't also supply audio, add an audio device input
 to the session

        if (![videoDevice hasMediaType:QTMediaTypeSound] && ![videoDevice
hasMediaType:QTMediaTypeMuxed]) {

            QTCaptureDevice *audioDevice = [QTCaptureDevice
defaultInputDeviceWithMediaType:QTMediaTypeSound];
            success = [audioDevice open:&error];

            if (!success) {
                audioDevice = nil;
                // Handle error
            }

            if (audioDevice) {
                mCaptureAudioDeviceInput = [[QTCaptureDeviceInput alloc]
initWithDevice:audioDevice];
```

```
                success = [mCaptureSession addInput:mCaptureAudioDeviceInput
error:&error];
                if (!success) {
                    // Handle error
                }
            }
        }

// Create the movie file output and add it to the session

        mCaptureMovieFileOutput = [[QTCaptureMovieFileOutput alloc] init];
        success = [mCaptureSession addOutput:mCaptureMovieFileOutput
error:&error];
        if (!success) {
            // Handle error
        }

        [mCaptureMovieFileOutput setDelegate:self];

// Associate the capture view in the UI with the session

        [mCaptureView setCaptureSession:mCaptureSession];

        [mCaptureSession startRunning];
    }

}

// Handle window closing notifications for your device input

- (void)windowWillClose:(NSNotification *)notification
{

    [mCaptureSession stopRunning];

    if ([[mCaptureVideoDeviceInput device] isOpen])
        [[mCaptureVideoDeviceInput device] close];

    if ([[mCaptureAudioDeviceInput device] isOpen])
        [[mCaptureAudioDeviceInput device] close];

}

// Handle deallocation of memory for your capture objects

- (void)dealloc
{
    [mCaptureSession release];
    [mCaptureVideoDeviceInput release];
    [mCaptureAudioDeviceInput release];
    [mCaptureMovieFileOutput release];

    [super dealloc];
}
```

4. Add these start and stop recording actions, and specify the output destination for your recorded media. The output is a QuickTime movie.

```
- (IBAction)startRecording:(id)sender
{
    [mCaptureMovieFileOutput recordToOutputFileURL:[NSURL
fileURLWithPath:@"/Users/Shared/My Recorded Movie.mov"]];
}

- (IBAction)stopRecording:(id)sender
{
    [mCaptureMovieFileOutput recordToOutputFileURL:nil];
}

// Do something with your QuickTime movie at the path you've specified at
/Users/Shared/My Recorded Movie.mov"

- (void)captureOutput:(QTCaptureFileOutput *)captureOutput
didFinishRecordingToOutputFileAtURL:(NSURL *)outputFileURL forConnections:(NSArray
 *)connections dueToError:(NSError *)error
{
    [[NSWorkspace sharedWorkspace] openURL:outputFileURL];
}


@end
```

Now you’re ready to build and compile your QTKit capture application. Once you’ve launched the application, you can begin to capture audio from your iSight camera or audio/video from a DV camera. The output is again recorded as a QuickTime movie, and then automatically opened in QuickTime Player on your desktop.

In the next chapter you’ll take on another coding assignment, this time creating a QTKit capture application that enables you to grab single frames from a video stream and output those frames, with great accuracy and reliability (avoiding tearing, for example), into a QuickTime movie. You’ll work with a technique that is common in the movie and TV industries, namely, stop or still motion animation.

Creating a QTKit Stop or Still Motion Application

Now that you've worked through the examples in the previous chapters of this guide—building and extending the functionality of your QTKit capture player application, adding audio and DV camera support—you'll be ready to take on another coding assignment. The goal here is, once again, to extend your knowledge of the QTKit capture API.

Because the API supports frame-accurate, real-time motion capture, you'll discover a broad range of possible uses and applications. One such usage is **still or stop motion animation**. This is a popular animation technique first introduced in Walt Disney's 1959 classic film *Noah's Ark*. Basically, it involves making still objects appear as if they are in motion by adding single frames together into a movie. It's a way of animating objects and bringing them visually to life on the screen. Stop or still motion animators have employed this technique—making static objects appear to move when played back at normal speed—in countless movies, TV commercials, and TV shows since the days of Disney.

Following the steps outlined in this chapter, you'll construct a simple still motion capture application that lets you capture a live video feed, grab frames one at a time with great accuracy, and then record the output of those frames to a QuickTime movie. You'll be able to accomplish this with less than 100 lines of Objective-C code, constructing the sample as you've done in previous chapters, in Xcode 3 and Interface Builder 3.

In building your still motion capture application, you'll work with the following three QTKit classes:

- `QTCaptureSession`. The primary interface for capturing media streams.
- `QTCaptureDecompressedVideoOutput`. The output destination for a `QTCaptureSession` object that you can use to process decompressed frames from the video that is being captured. Using the methods provided in this class, you can produce decompressed video frames that are suitable for high-quality video processing.
- `QTCaptureDeviceInput`. The input source for media devices, such as cameras and microphones.

Set Up Your Project

If you've come to this chapter without working through the code examples in the previous chapters of this guide, you may want to return to the sections beginning with "First Steps" (page 17). Those sections provide you with a background understanding of how to work with Xcode and Interface Builder.

You'll need, as described in previous chapters, to be running Mac OS X v10.5 and have the following items installed on your system:

- Xcode 3 and Interface Builder 3. Note that although you can use Xcode 2.2 to build your project, to take full advantage of the new programming features available, you'll want to use Xcode 3. Also, Interface Builder 3 provides a new paradigm for working with and building the user interface for your application. Palettes are no longer provided; instead, you'll work with a new library of controls that are designed to enable you to hook up the components of your user interface with greater ease and efficiency. In the end, you'll be able to build applications faster and take advantage of this rich, new library of controls.

- The QuickTime Kit framework, which resides in the Mac OS X v10.5 `System/Library/Frameworks` directory as `QTKit.framework`.
- An iSight camera connected to your computer.

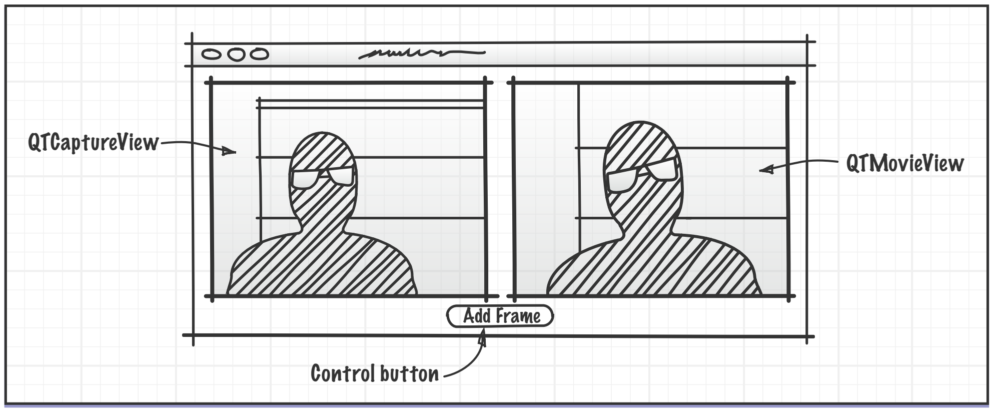
Important: The complete sample code discussed in this chapter is available for download at Sample Code: QuickTime Cocoa, as `StillMotion`.

Note: The `StillMotion` sample code described in this chapter will not support input from DV cameras, which are of type `QTMediaTypeMuxed`, rather than `QTMediaTypeVideo`. To add support for DV cameras, read the chapter "Adding Audio Input and DV Camera Support" (page 37).

Prototype the Still Motion Capture Application

Just as you've done in the section "Prototype the Capture Player" (page 18), you may want to start by creating a rough sketch of your QTKit still motion capture application. Think, again, of what design elements you want to incorporate into the application. Rather than simply jumping into Interface Builder and doing your prototype there, you may want to visualize the elements first in your rough sketch, as shown in Figure 4-2.

Figure 4-1 Prototype sketch of QTKit still motion capture application



In this design prototype, you can start with three simple objects: a capture view, a QuickTime movie view, and a single button to add frames. These will be the building blocks for your application. You can add more complexity to the design later on. After you've sketched out your prototype, think how you'll be able to hook up the objects in Interface Builder and what code you need in your Xcode project to make this happen.

Create the Project Using Xcode 3

To create the project, follow these steps:

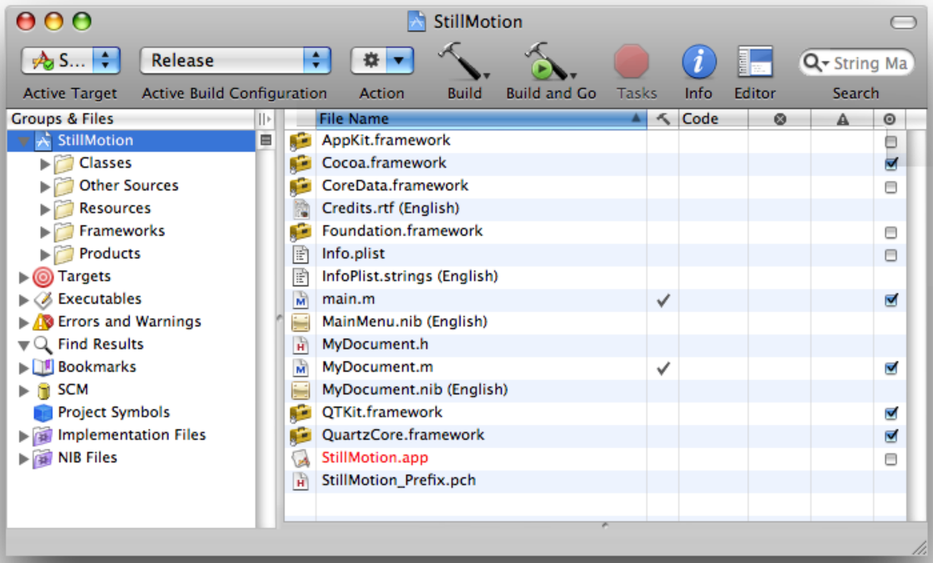
1. Launch Xcode 3 (shown in Figure 4-2) and choose File > New Project.

Figure 4-2 The Xcode 3 icon

2. When the new project window appears, select Cocoa Document-based Application.

3. Name the project `StillMotion` and navigate to the location where you want the Xcode application to create the project folder. Now the Xcode project window appears, as shown in Figure 4-3.

Figure 4-3 The StillMotion Xcode project window



4. Next, you need to add the QuickTime Kit framework to your `StillMotion` project. This framework resides in the `System/Library/Frameworks` directory. Although obvious, this step is sometimes easy to forget. Note that you don't need to add the QuickTime framework to your project, just the QuickTime Kit framework. Choose Project > Add to Project.

5. Select `QTKit.framework`, and click Add when the Add To Targets window appears to add it to your project.

6. Now you also need to add the Quartz Core framework to your project. It resides in the `System/Library/Frameworks` directory. Choose Project > Add to Project. Select `QuartzCore.framework`, and click Add when the Add To Targets window appears.

Important: This completes the first sequence of steps in your project. In the next sequence, you'll move ahead to define actions and outlets in Xcode *before* working with Interface Builder. This may involve something of a paradigm shift in how you may be used to building and constructing an application with versions of Interface Builder prior to Interface Builder 3. Because you've already prototyped your QTKit still motion capture application, at least in rough form with a clearly defined data model, you can now determine which actions and outlets need to be implemented. In this case, you have a `QTCaptureView` object, which is a subclass of NSView, a `QTMovieView` object to display your captured frames and one button to record your captured media content and add each single frame to your QuickTime movie output.

Import the QTKit Headers and Set Up Your Implementation File

1. Double-click your `MyDocument.h` declaration file in your Xcode project and open it. In the file, delete the `#import <Cocoa/Cocoa.h>` statement and replace it with `#import <QTKit/QTkit.h>`.

2. Double-click your `MyDocument.m` implementation file in your project to open it. Delete the contents of the file except for the following lines of code:

```
#import "MyDocument.h"
@implementation MyDocument
- (NSString *)windowNibName
{
    return @"MyDocument";
}
@end
```

Determine the Actions and Outlets You Want

1. Now you can begin adding outlets and actions. In your `MyDocument.h` file, add the instance variables `mCaptureView` and `mMovieView` in the following lines of code:

```
IBOutlet QTCaptureView *mCaptureView;
IBOutlet QTMovieView   *mMovieView;
```

2. You also want to add this action method:

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

3. Now open your `MyDocument.m` file and add the same action method, followed by braces for the code you'll add later to implement this action.

```
- (IBAction)addFrame:(id)sender
{
}
```

This completes the second stage of your project. Now you'll need to shift gears and work with Interface Builder 3 to construct the user interface for your project.

Create the User Interface Using Interface Builder 3

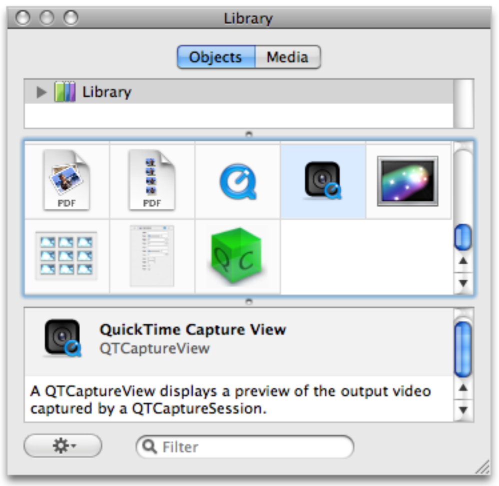
In the next phase of your project you'll see how seamlessly Interface Builder and Xcode work together, enabling you to construct and implement the various elements in your project more efficiently and with less overhead.

1. Open Interface Builder 3 (Figure 2-4) and click the `MyDocument.nib` file in your Xcode project window. Because of the new integration between Xcode 3 and Interface Builder 3, you'll find the actions and outlets you've declared in your `MyDocument.h` file are also synchronously updated in Interface Builder 3. This will become apparent once you begin to work with your MyDocument nib file and the library of controls available in Interface Builder 3.

   Figure 4-4 The new Interface Builder 3 icon

2. In Interface Builder 3, you'll find a new library of controls. Scroll down until you find the QuickTime Capture View control, as shown in Figure 4-5.

   Figure 4-5 The QuickTime Capture View object in the library

   QuickTime Capture View
   QTCaptureView

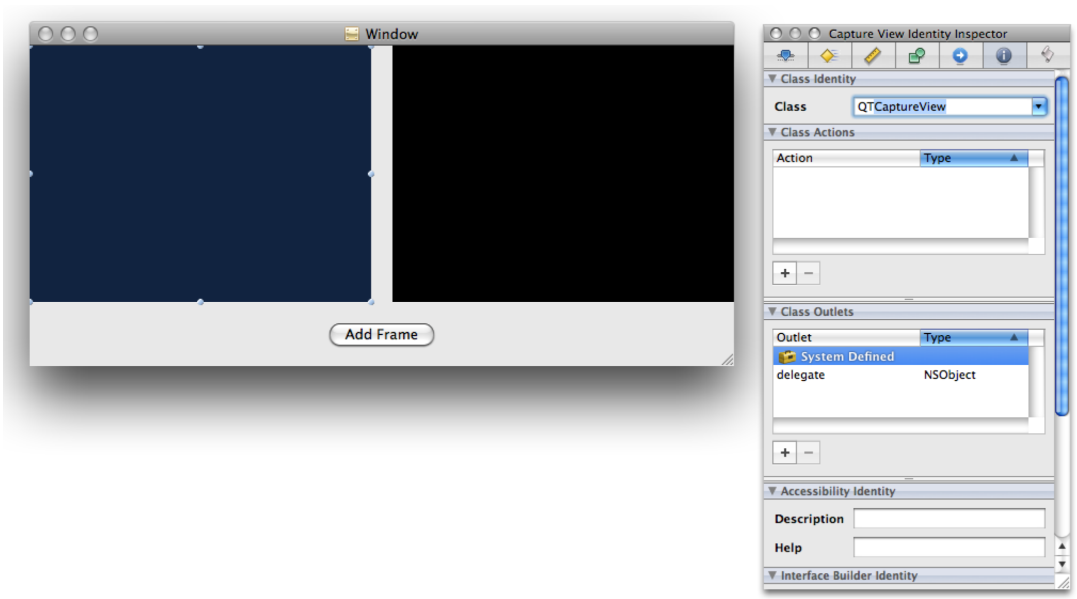
   A QTCaptureView displays a preview of the output video captured by a QTCaptureSession.

   The `QTCaptureView` object provides you with an instance of a view subclass to display a preview of the video output that is captured by a capture session.

3. Drag the `QTCaptureView` object into your window and resize the object to fit the window, allowing room at the bottom for your Add Frame button (already shown in the illustration below) and to the right for your `QTMovieView` object in your QTKit still motion capture application.

Choose Tools > Inspector. In the Identity Inspector, select the information ("i") icon. Click in the field Class and your `QTCaptureView` object appears, as shown in Figure 4-6

Figure 4-6 The QTCaptureView window and Class Identity in the Inspector

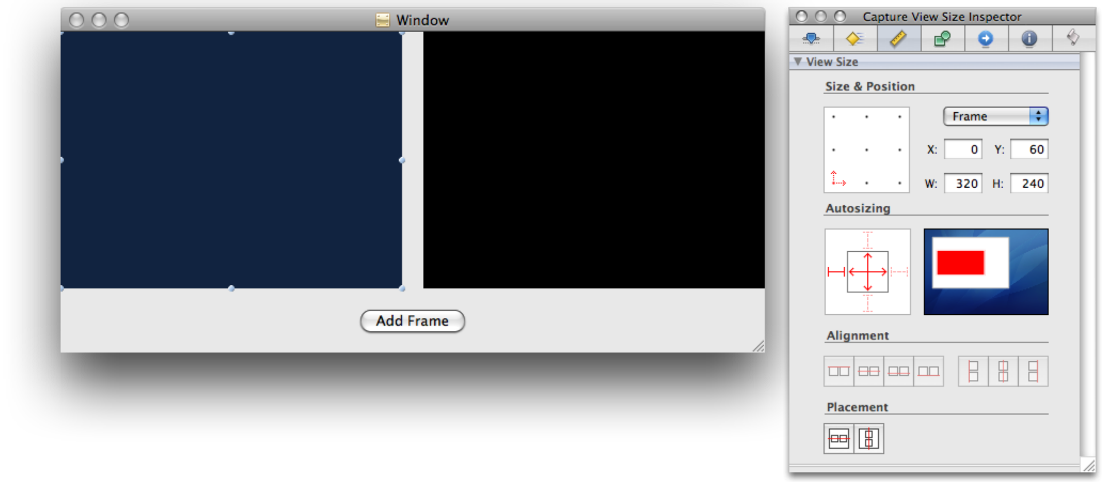


Class: QTCaptureView

Outlet	Type
delegate	NSObject

4. Set the autosizing for the object in the Capture View Size Inspector, as shown in Figure 2-7.

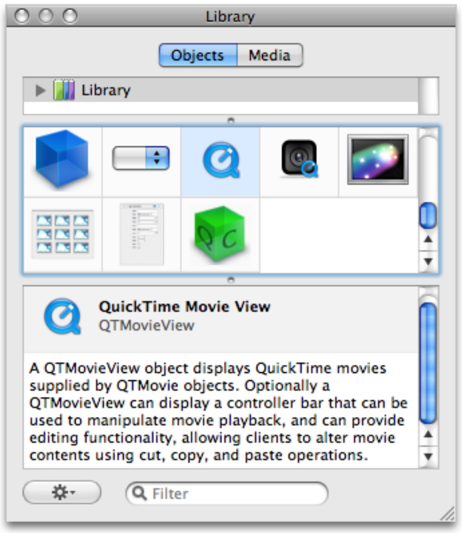
Figure 4-7 Setting the autosizing for your `QTCaptureView` object



X: 0 Y: 60
W: 320 H: 240

5. Now you want to repeat the same sequence of steps to add your `QTMovieView` object to your Window (already shown above). Scroll down in the Library of controls until you find the `QTMovieView` object, shown in Figure 4-8.

Figure 4-8 The QTMovieView control in the Interface Builder library



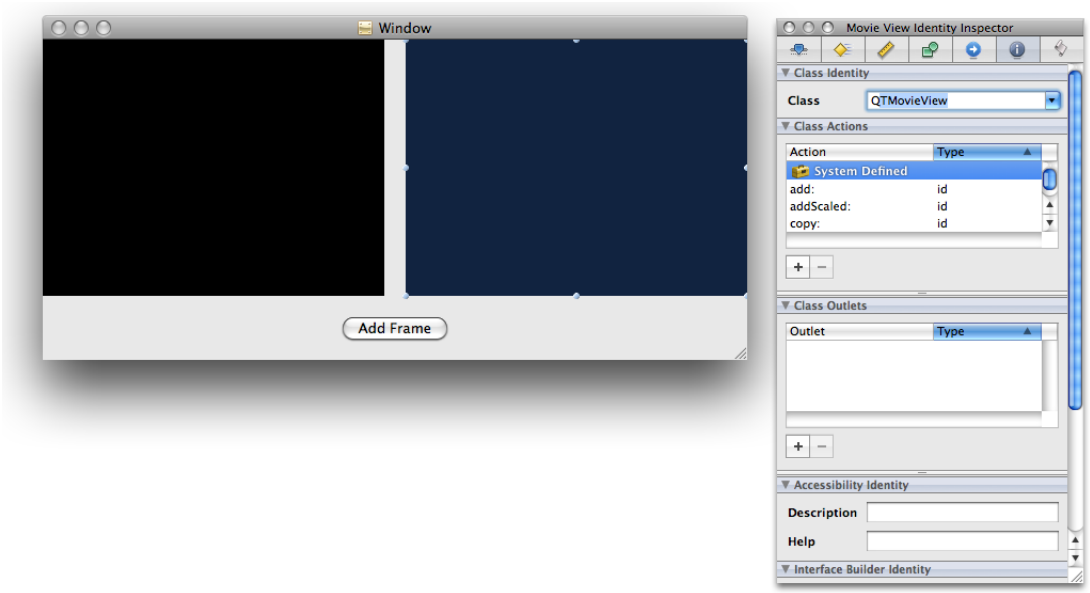
QuickTime Movie View
QTMovieView

A QTMovieView object displays QuickTime movies supplied by QTMovie objects. Optionally a QTMovieView can display a controller bar that can be used to manipulate movie playback, and can provide editing functionality, allowing clients to alter movie contents using cut, copy, and paste operations.

6. Select the `QTMovieView` object (symbolized by the blue Q) and drag it into your Window next to the `QTCaptureView` object, shown below.

7. Choose Tools > Inspector. In the Identity Inspector, select the information ("i") icon. Click in the Class field and your `QTMovieView` object appears, as shown in Figure 4-9.

Figure 4-9 The QTMovieView object defined in the field Class



Class: QTMovieView

Action	Type
System Defined	
add:	id
addScaled:	id
copy:	id

Outlet	Type

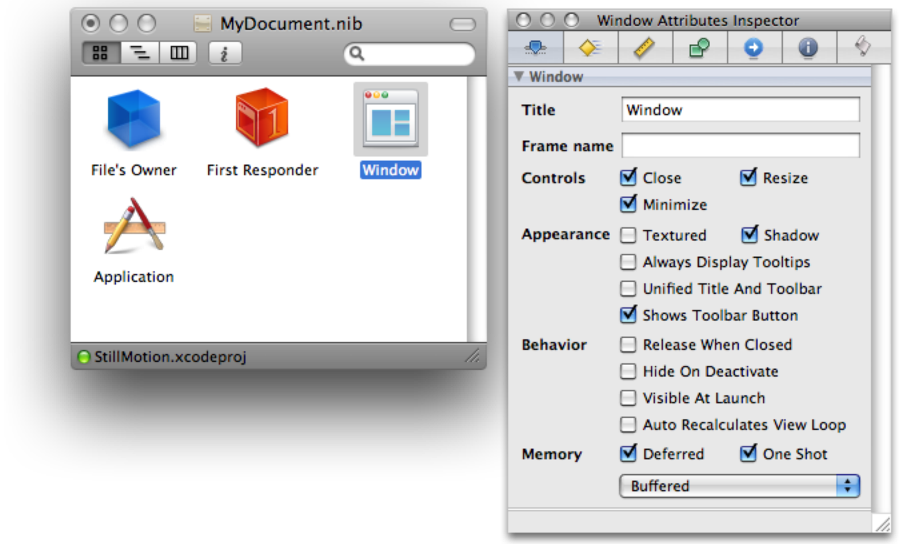
Description:

Help:

8. Follow the procedure in step 4 above to set the autosizing for your `QTMovieView` object.

9. Now you want to specify the Window attributes in your `MyDocument.nib` file. Select the Window object in your nib and click the attributes icon in the Inspector, as shown in Figure 4-10.

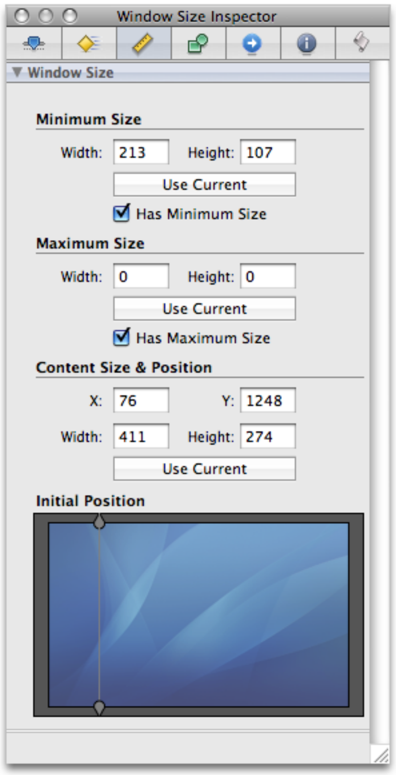
Figure 4-10 The Window attributes in `MyDocument.nib`



Title: Window
Frame name:
Controls: ✓ Close ✓ Resize ✓ Minimize
Appearance: ☐ Textured ✓ Shadow ☐ Always Display Tooltips ☐ Unified Title And Toolbar ✓ Shows Toolbar Button
Behavior: ☐ Release When Closed ☐ Hide On Deactivate ☐ Visible At Launch ☐ Auto Recalculates View Loop
Memory: ✓ Deferred ✓ One Shot
Buffered

10. Define the Window size you want in your `MyDocument.nib` by selecting the size icon (symbolized by a ruler) in the Window Inspector, shown in Figure 4-11.

Figure 4-11 Defining the size of the Window


Window Size Inspector

Window Size

Minimum Size
Width: 213 Height: 107
Use Current
☑ Has Minimum Size

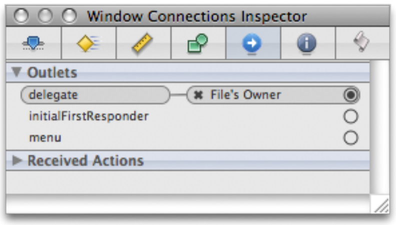
Maximum Size
Width: 0 Height: 0
Use Current
☑ Has Maximum Size

Content Size & Position
X: 76 Y: 1248
Width: 411 Height: 274
Use Current

Initial Position

11. Specify the delegate outlet connections of File's Owner in the Window Connections Inspector, as shown in Figure 4-12.

Figure 4-12 Specifying the delegate outlet connection for File's Owner

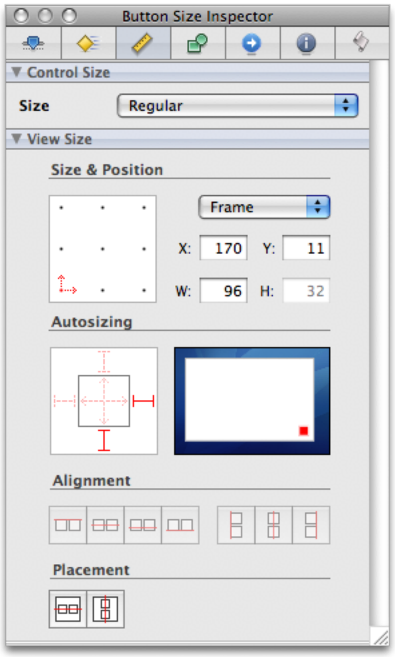

Window Connections Inspector

Outlets
delegate — File's Owner ●
initialFirstResponder ○
menu ○

Received Actions

12. In the Library, select the Push Button control and drag it to the Window. Enter the text **Add Frame**. In autosizing, set the struts for the button at the center and right outside corner, leaving the inside struts untouched, as shown in Figure 4-13.

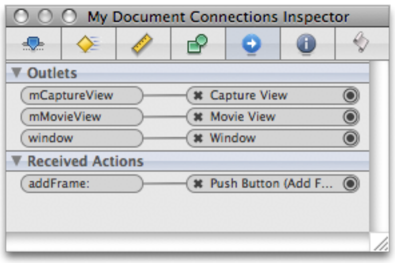
Figure 4-13 Specifying the autosizing for the Add Frame button



Size: Regular
Frame
X: 170 Y: 11
W: 96 H: 32

13. Select the `MyDocument.nib` and click the Connections Inspector. Now you want to wire up the outlets and received actions, as shown in Figure 4-14. Control-drag each outlet instance variable to the appropriate `MyDocument.nib` object.

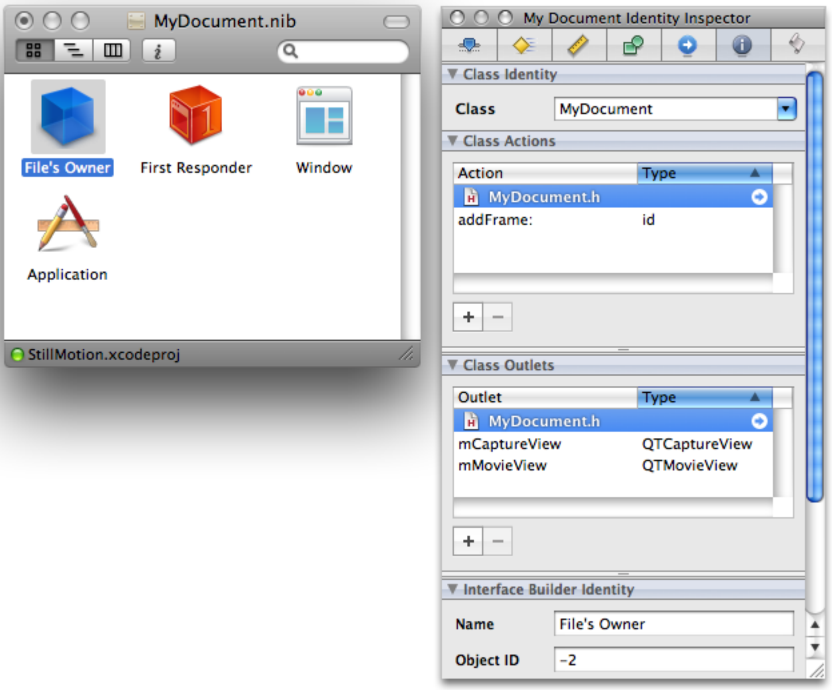
Figure 4-14 Specifiying the actions and outlet connections in MyDocument.nib



mCaptureView — Capture View
mMovieView — Movie View
window — Window
addFrame: — Push Button (Add F...

14. Select the File's Owner object in your `MyDocument.nib` and then click the Class Identity icon in the Interface Builder Inspector, as shown in Figure 4-15. Note that the green light at the left corner of your StillMotion.xcodeproj is turned on, indicating that Xcode and Interface Builder have synchronized the actions and outlets in your project.

Figure 4-15 The My Document Identity Inspector with File's Owner selected



Class: MyDocument

Action	Type
MyDocument.h	
addFrame:	id

Outlet	Type
MyDocument.h	
mCaptureView	QTCaptureView
mMovieView	QTMovieView

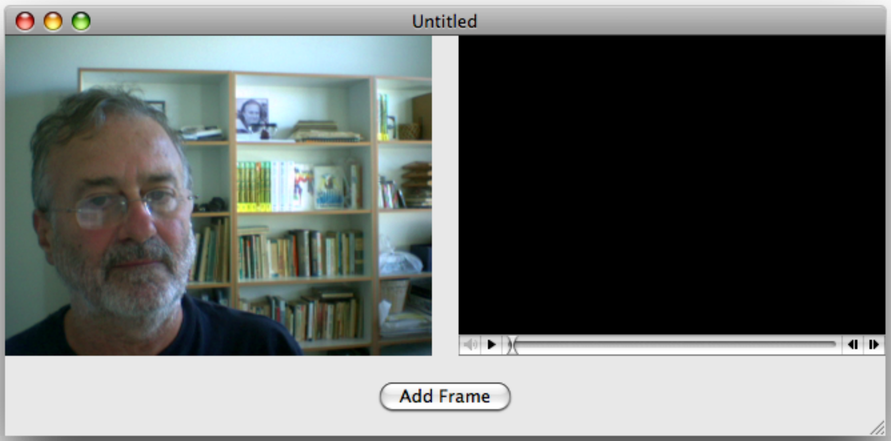
Name: File's Owner

Object ID: -2

Prepare to Capture Single Frame Video

In the last phase of your project, you've seen how seamlessly Interface Builder and Xcode work together, enabling you to construct and implement the various elements in your project more efficiently and with less overhead. After completing the code you need to add your Xcode project––described in the next section, "Complete the Project Nib File in Xcode" (page 52)––you'll be ready to move ahead and capture single frame video, using your still motion capture application, as shown in Figure 4-16.

Figure 4-16 Preparing to capture single video frames and outputting those frames to a QuickTime movie



Complete the Project Nib File in Xcode

To complete the project nib file, you'll need to define the instance variables that point to the capture session, as well as to the device input and decompressed video output objects.

1. In your Xcode project, you need to add the instance variables to the interface declaration. Add these lines of code in your `MyDocument.h` declaration file:

```
@interface MyDocument : NSDocument
{
   QTMovie                           *mMovie
   QTCaptureSession                  *mCaptureSession;
   QTCaptureDeviceInput              *mCaptureDeviceInput;
   QTCaptureDecompressedVideoOutput  *mCaptureDecompressedVideoOutput;
}
```

   The `mMovie` instance variable points to the `QTMovie` object while the `mCaptureSession` instance variable points to the `QTCaptureSession` object. Likewise, the `*mCaptureDeviceInput` instance variable points to the `QTCaptureDeviceInput` object while the next line declares that the `mCaptureDecompressedVideoOutput` instance variable points to the `QTCaptureDecompressedVideoOutput` object.

2. There is one more instance variable you need to declare in this file: `mCurrentImageBuffer`. This instance variable stores the most recent frame that you've grabbed in a `CVImageBufferRef`. Add this line of code, following your last declaration:

```
CVImageBufferRef                  mCurrentImageBuffer;
```

3. That completes the code you need to add to your `MyDocument.h` file. Now you want to open your `MyDocument.m` file and prepare to add the following blocks of code to your project. Note that the code is commented for better understanding and comprehension.

Important: There is a specific, though not necessarily rigid, order of steps you want to follow in constructing your code. Think of these as specific tasks you want to accomplish in your project.

- **a.** Create an empty movie that writes to mutable data in memory, using the `initToWritableData:` method.
- **b.** Set up a capture session that outputs the raw frames you want to grab.
- **c.** Find a video device and add a device input for that device to the capture session.
- **d.** Add a decompressed video output that returns the raw frames you've grabbed to the session and then previews the video from the session in the document window.
- **e.** Start the session, using the `startRunning` method you've used previously in the `MyRecorder` sample code.
- **f.** Call a delegate method whenever the `QTCaptureDecompressedVideoOutput` object receives a frame.
- **g.** Store the latest frame. Do this in a `@synchronized` block because the delegate method is not called on the main thread.
- **h.** Get the most recent frame. Do this in a `@synchronized` block because the delegate method that sets the most recent frame is not called on the main thread.
- **i.** Create an `NSImage` and add it to the movie.

4. Following the steps outlined above, add this block of code to your `MyDocument.m` file.

```
- (void)windowControllerDidLoadNib:(NSWindowController *) aController
{
    NSError *error = nil;
    [super windowControllerDidLoadNib:aController];
    [[aController window] setDelegate:self];
    if (!mMovie) {
        // Create an empty movie that writes to mutable data in memory
       mMovie = [[QTMovie alloc] initToWritableData:[NSMutableData data] error:&error];
        if (!mMovie) {
            [[NSAlert alertWithError:error] runModal];
            return;
        }
    }
    [mMovieView setMovie:mMovie];
    if (!mCaptureSession) {
        // Set up a capture session that outputs raw frames
        BOOL success;
        mCaptureSession = [[QTCaptureSession alloc] init];
        // Find a video device
        QTCaptureDevice *device = [QTCaptureDevice
defaultInputDeviceWithMediaType:QTMediaTypeVideo];
        success = [device open:&error];
        if (!success) {
            [[NSAlert alertWithError:error] runModal];
            return;
        }
        // Add a device input for that device to the capture session
        mCaptureDeviceInput = [[QTCaptureDeviceInput alloc] initWithDevice:device];
        success = [mCaptureSession addInput:mCaptureDeviceInput error:&error];
        if (!success) {
            [[NSAlert alertWithError:error] runModal];
            return;
        }
        // Add a decompressed video output that returns raw frames to the session
        mCaptureDecompressedVideoOutput = [[QTCaptureDecompressedVideoOutput alloc]
init];
        [mCaptureDecompressedVideoOutput setDelegate:self];
        success = [mCaptureSession addOutput:mCaptureDecompressedVideoOutput
error:&error];
        if (!success) {
            [[NSAlert alertWithError:error] runModal];
            return;
        }
        // Preview the video from the session in the document window
        [mCaptureView setCaptureSession:mCaptureSession];

        // Start the session
        [mCaptureSession startRunning];
    }
}
```

5. Add these lines to handle window closing notifications for your device input and stop the capture session:

```
- (void)windowWillClose:(NSNotification *)notification
{
    [mCaptureSession stopRunning];
    QTCaptureDevice *device = [mCaptureDeviceInput device];
    if ([device isOpen])
        [device close];
}
```

6. Insert the following block of code to handle deallocation of memory for your capture objects:

```
- (void)dealloc
{
    [mMovie release];
    [mCaptureSession release];
    [mCaptureDeviceInput release];
    [mCaptureDecompressedVideoOutput release];
    [super dealloc];
}
```

7. Add the following lines of code to specify the output destination for your recorded media, in this case an editable QuickTime movie:

```
- (BOOL)readFromURL:(NSURL *)absoluteURL ofType:(NSString *)typeName
error:(NSError **)outError
{
   QTMovie *newMovie = [[QTMovie alloc] initWithURL:absoluteURL error:outError];
    if (newMovie) {
        [newMovie setAttribute:[NSNumber numberWithBool:YES]
forKey:QTMovieEditableAttribute];
        [mMovie release];
        mMovie = newMovie;
    }
    return (newMovie != nil);
}
- (BOOL)writeToURL:(NSURL *)absoluteURL ofType:(NSString *)typeName error:(NSError
 **)outError
{
    return [mMovie writeToFile:[absoluteURL path] withAttributes:[NSDictionary
 dictionaryWithObject:[NSNumber numberWithBool:YES] forKey:QTMovieFlatten]
error:outError];
}
- (BOOL)writeToURL:(NSURL *)absoluteURL ofType:(NSString *)typeName error:(NSError
 **)outError
{
    return [mMovie writeToFile:[absoluteURL path] withAttributes:[NSDictionary
 dictionaryWithObject:[NSNumber numberWithBool:YES] forKey:QTMovieFlatten]
error:outError];
}
```

8. Add these lines of code to call a delegate method whenever the `QTCaptureDecompressedVideoOutput` object receives a frame:

```
 - (void)captureOutput:(QTCaptureOutput *)captureOutput
didOutputVideoFrame:(CVImageBufferRef)videoFrame withSampleBuffer:(QTSampleBuffer
 *)sampleBuffer fromConnection:(QTCaptureConnection *)connection
{
    // Store the latest frame
    // This must be done in a @synchronized block because this delegate method
 is not called on the main thread
    CVImageBufferRef imageBufferToRelease;

    CVBufferRetain(videoFrame);

    @synchronized (self) {
        imageBufferToRelease = mCurrentImageBuffer;
        mCurrentImageBuffer = videoFrame;
    }
    CVBufferRelease(imageBufferToRelease);
}
```

9. Now you want to specify the `addFrame:` action method and get the most recent frame that you've grabbed. Do this in a `@synchronized` block because the delegate method that sets the most recent frame is not called on the main thread. Note that you're wrapping a `CVImageBufferRef` object into an `NSImage`. After you create an `NSImage`, you can then add it to the movie.

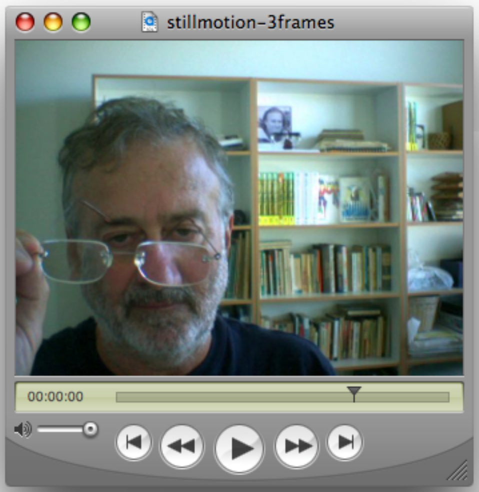
```
- (IBAction)addFrame:(id)sender
{
    CVImageBufferRef imageBuffer;
    @synchronized (self) {
        imageBuffer = CVBufferRetain(mCurrentImageBuffer);
    }
    if (imageBuffer) {
        // Create an NSImage and add it to the movie
        NSCIImageRep *imageRep = [NSCIImageRep imageRepWithCIImage:[CIImage
imageWithCVImageBuffer:imageBuffer]];
        NSImage *image = [[[NSImage alloc] initWithSize:[imageRep size]]
autorelease];
        [image addRepresentation:imageRep];
        CVBufferRelease(imageBuffer);
        [mMovie addImage:image forDuration:QTMakeTime(1, 10)
withAttributes:[NSDictionary dictionaryWithObjectsAndKeys:
@"jpeg", QTAddImageCodecType, nil]];
        [mMovie setCurrentTime:[mMovie duration]];
        [mMovieView setNeedsDisplay:YES];
        [self updateChangeCount:NSChangeDone];
    }
}
```

Implement and Build Your Still Motion Capture Application

After you've saved your project, click Build and Go. After compiling, click the **Add Frame** button to record each captured frame and output that frame to a QuickTime movie. The output of your captured session is saved as a QuickTime movie .

Now you can begin capturing and recording with your QTKit still motion capture player application. Using a simple iSight camera, your output appears as a three-frame QuickTime movie, as shown in Figure 4-17.

Figure 4-17 Still motion recorded output as a three-frame QuickTime movie



Document Revision History

This table describes the changes to *QTKit Capture Programming Guide*.

Date	Notes
2007-10-31	Corrected an error in chapter on creating a QTKit stop or still motion application; also fixed table formatting.
2007-07-24	New document that describes how to capture media and output it to QuickTime movies.