



LEARNING AV Foundation

A Hands-on Guide to Mastering the AV Foundation Framework



BOB McCUNE

Foreword by **CHRIS ADAMSON**, author of *Learning Core Audio*

About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Learning AV Foundation

**A Hands-on Guide to
Mastering the AV Foundation
Framework**

Bob McCune

 Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2014944245

Copyright © 2015 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-96180-8

ISBN-10: 0-321-96180-3

Text printed in the United States on recycled paper at Courier in Westford,

Massachusetts.

First printing: October 2014

Editor-in-Chief

Mark Taub

Senior Acquisitions Editor

Trina MacDonald

Development Editor

Chris Zahn

Managing Editor

Kristy Hart

Project Editor

Elaine Wiley

Copy Editor

Barbara Hacha

Senior Indexer

Cheryl Lenser

Proofreader

Katherine Matejka

Technical Reviewers

Chris Adamson

Ryder Mackay

Jon Steinmetz

Editorial Assistant

Olivia Basegio

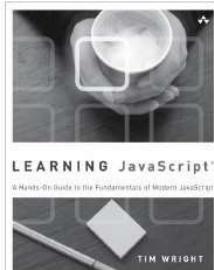
Cover Designer

Chuti Prasertsith

Senior Compositor

Gloria Schurick

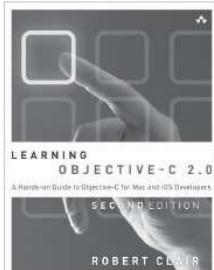
Addison-Wesley Learning Series



LEARNING JavaScript™

A Hands-On Guide to the Fundamentals of Modern JavaScript

TIM WRIGHT

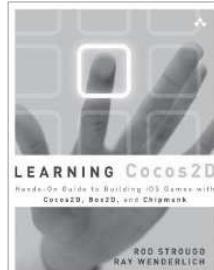


LEARNING
OBJECTIVE-C 2.0

A Hands-On Guide to Objective-C for Mac and iOS Developers

SECOND EDITION

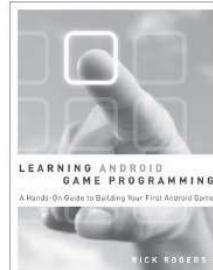
ROBERT CLAIR



LEARNING Cocos2D

Hands-On Guide to Building iOS Games with Cocos2D, Box2D, and Chipmunk

ROD STROUD
RAY WENDERLICH



LEARNING ANDROID
GAME PROGRAMMING

A Hands-On Guide to Building Your First Android Game

RICK ROGERS

▼ Addison-Wesley

Visit informit.com/learningseries for a complete list of available publications.

The Addison-Wesley Learning Series is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

▼ Addison-Wesley

informIT®

Safari®
Books Online

ALWAYS LEARNING

PEARSON



*I dedicate this book to my loving wife, Linda,
who supports me in all my crazy endeavors.*



Contents

Preface

Part I AV Foundation Essentials

1 Getting Started with AV Foundation

What Is AV Foundation?

Where Does AV Foundation Fit?

Decomposing AV Foundation

Understanding Digital Media

Digital Media Compression

Container Formats

Hello AV Foundation

Summary

Challenge

2 Playing and Recording Audio

Mac and iOS Audio Environments

Understanding Audio Sessions

Audio Playback with AVAudioPlayer

Building an Audio Looper

Configuring the Audio Session

Handling Interruptions

Responding to Route Changes

Audio Recording with AVAudioRecorder

Building a Voice Memo App

Enabling Audio Metering

Summary

3 Working with Assets and Metadata

[Understanding Assets](#)

[Creating an Asset](#)

[Asynchronous Loading](#)

[Media Metadata](#)

[Working with Metadata](#)

[Building the MetaManager App](#)

[Saving Metadata](#)

[Summary](#)

[Challenge](#)

4 Playing Video

[Playback Overview](#)

[Playback Recipe](#)

[Working with Time](#)

[Building a Video Player](#)

[Time Observation](#)

[Creating a Visual Scrubber](#)

[Showing Subtitles](#)

[Airplay](#)

[Summary](#)

[Challenge](#)

5 Using AV Kit

[AV Kit for iOS](#)

[AV Kit for Mac OS X](#)

[First Steps](#)

[Control Styles](#)

[Going Further](#)

[Working with Chapters](#)

[Enabling Trimming](#)

[Exporting](#)

[Movie Modernization](#)

[Summary](#)

[Challenge](#)

Part II Media Capture

6 Capturing Media

[Capture Overview](#)

[Simple Recipe](#)

[Building a Camera App](#)

[Summary](#)

[Challenge](#)

7 Using Advanced Capture Features

[Video Zooming](#)

[Face Detection](#)

[Machine-Readable Code Detection](#)

[Using High Frame Rate Capture](#)

[Processing Video](#)

[Understanding CMSampleBuffer](#)

[Summary](#)

[Challenge](#)

8 Reading and Writing Media

[Overview](#)

[Building an Audio Waveform View](#)

[Advanced Capture Recording](#)

[Summary](#)

[Challenge](#)

Part III Media Creation and Editing

9 Composing and Editing Media

[Composing Media](#)

[Working with Time](#)

[Basic Recipe](#)

[Introducing 15 Seconds](#)

[Building a Composition](#)

[Exporting the Composition](#)

[Summary](#)

[Challenge](#)

[10 Mixing Audio](#)

[Mixing Audio](#)

[Mixing Audio in the 15 Seconds App](#)

[Summary](#)

[Challenge](#)

[11 Building Video Transitions](#)

[Overview](#)

[Conceptual Steps](#)

[15 Seconds: Adding Video Transitions](#)

[Summary](#)

[Challenge](#)

[12 Layering Animated Content](#)

[Using Core Animation](#)

[Using Core Animation with AV Foundation](#)

[15 Seconds: Adding Animated Titles](#)

[Preparing the Composition](#)

[Summary](#)

[Challenge](#)

Foreword

Yeah, we knew QuickTime's goose was cooked.

It had served us well for two decades, but that's the problem. Apple's essential media framework was a product of the late 1980s. By the mid 2000s, it had accumulated plenty of cruft: old programming practices, dependencies on system APIs that had fallen out of favor, and features that didn't stand the test of time (wired sprites, anyone?). Heck, it preferred big-endian numeric values, because that's what the Motorola 68000 series CPUs used. That's right, QuickTime had already made two CPU transitions: from 680x0 to PowerPC, and then again to Intel x86.

And QuickTime's evolution in the first decade of the new century was hard to make sense of. Apple built a Java wrapper around QuickTime, then updated it again and left out half the features. They did the same incomplete job in Objective-C and called it QTKit. And then there was a Windows version of QuickTime that stopped getting meaningful updates, and nobody at Apple would tell us why.

Usually with Apple, that means they're up to something.

What they were up to was the iPhone, of course. But the first SDK we got for iPhone shipped with a minimum of media support: a full-screen video player that took over your application and the low-level Core Audio library. There was an obvious, enormous hole in the media software stack for what Apple insisted was "the best iPod we've ever made." Yet we knew they couldn't port QuickTime over to the iPhone, considering they were already walking away from it on the desktop.

Oh yeah, they were up to something.

Bits and pieces of new media functionality on iPhone popped up here and there over the next few years: a "Media Player" framework to let us query and play the music library songs, and some Objective-C wrappers around Core Audio's Audio Queue, so that playing from or recording to a file was no longer a 200-line exercise in drudgery. These latter classes were curiously assigned to a new framework—"AV Foundation"—which seemed a misnomer in the iPhone OS 3 era, when it was all "A" and no "V."

In retrospect, we really should have known they were up to something.

Then, in 2010, Apple was finally ready to show us what they'd been up to all this time. Apple's Meriko Borogove stood up at the WWDC "Graphics State of the Union," showed off iMovie for iPhone, and said that everything Apple used to make this video editor was now available to iOS developers. AV Foundation, formerly consisting of those odd little Core Audio wrappers, was now 40 classes of audio-video processing power. Capture, editing, playback, and export—pretty much everything we ever actually did with QuickTime (sorry, wired sprites)—were all present and accounted for.

And now they fit in your pocket or purse.

Relieved of 1990s legacies, the new classes were products of genuinely modern thinking. On iOS, they were among the first to make use of Objective-C blocks to handle asynchronous concerns like lengthy media export, practices that now seem like second nature to iOS developers. Back on the Mac, a few of us looked enviously to iOS, given that our choices now consisted of 32-bit-only QuickTime with all its archaic bits or the bowdlerized QTKit. It was hardly surprising in OS X 10.7 ("Lion") when AV Foundation made its debut on the Mac, or in OS X 10.9 ("Mavericks") when QuickTime was formally deprecated in favor of AV Foundation.

Don't assume from this story that it's all rainbows and puppies for us, though. Media development is still a tricky business. We deal with huge amounts of data, razor-thin timing windows for real-time processing, and high user expectations when our stuff is literally the only thing they're looking at.

There's also a lot of material to understand: the sciences of acoustics and vision, solid programming practices, and the fact that AV Foundation brings in references to other frameworks, like Core Media, Core Video, Core Image, Core Audio, Media Player (on iOS), Video Toolbox (on OS X), and more. It's also not always easy to intuit an API where all the class names seem to have been created with those "poetry" refrigerator magnets, swapping around the terms "AV," "Composition," "Instruction," "Video," "Layer," and "Mutable" to create at least a dozen of the actual class names (we should award a prize to whoever can find the most). It doesn't really make sense that an `AVMutableComposition` and an `AVMutableVideoComposition` aren't formally related to each other at all.

Add to this the usual challenges of mastering Apple frameworks: the unstated assumptions, the offhand mentions of other frameworks and libraries, and the

references to sample code from a WWDC session three years ago that may no longer be available. And don't bother bookmarking anything; Apple reorganizes their developer website at least once a year, breaking all the external links.

Honestly, we have needed a proper book on AV Foundation for as long as it's been a public API.

The book you're reading now is the product of trial-and-error, digging through documentation and header files, scouring forums, and banging stuff off the compiler, the simulator, and the device until it works. It brings together the knowledge of many sources, and the expertise and experience of many developers, into a handy package. Many of us who've been leaning on AV Foundation for the past few years, pushing past the easy examples and figuring out what it's really capable of, have been happy to see Bob McCune take up the torch here, to enlighten AV Foundation developers with a singular guide to mastering this wide-reaching framework. Bob's been working on this for well over a year, exasperated like all of us when a search for information turns up one's own forum posts and blogs and not much else.

On Twitter, Bob and I joked that one session of our back-and-forth tweets could itself double the Google hit count for a term like

`AVVideoCompositionLayerInstruction`, inasmuch as such long-winded class names can even fit in a tweet. At one point, we joked that hashtag `#avfoundation` turned up about as much useful information as a nonsense hashtag like `#sidewaysbondagecake`. We need more of this information out there where people can see it, and Bob's done a terrific job here.

It's great to see this long, long project finally come to fruition. With more developers empowered to get the most out of AV Foundation, we may see a new surge of great audio and video applications on Apple platforms over the next few years. In 1990, we had postage-stamp sized videos with a tiny set of blotchy colors. Today, we shoot HD video on iPhones and send it to our 50" TVs over AirPlay without a second thought.

It's a fine time to join the ranks of AV Foundation developers. We can't wait to see what you do with your app when you're done with this book.

—Chris Adamson

Author of *Learning Core Audio* (Addison-Wesley Professional, 2012) and *QuickTime for Java: A Developer's Notebook* (O'Reilly Media, 2005)

August 2014

Preface

It's been inspiring to see the digital media revolution that's been underway over the past few years. The introduction of the iPhone and the rise of mobile computing in general, along with the availability of high-speed networks, has forever changed the way we create, consume, and share digital media.

Watching a video is no longer a passive activity relegated to our living rooms. Today, video is active and available on-demand everywhere we go. The ability to capture high-resolution, stylized photos isn't limited to professional photographers with high-end cameras and software, but is at the fingertips of everyone with an iOS device. Filmmakers and musicians who formerly could see their vision realized only in a professional studio can now do so on their laptops and mobile devices. The digital media revolution is underway, but it's really just getting started, and the technology at the heart of this revolution on iOS and OS X is AV Foundation.

I have been very happy to have the opportunity to write this book, because I believe it is one that is long overdue. AV Foundation powers so many of the top applications on the App Store, but it's a framework that is not well understood by the community at large. Learning to use AV Foundation can be challenging. It's a large and advanced framework with a broad set of features and capabilities. The AV Foundation Programming Guide, although improved over the past year, is still lacking and really just scratches the surface. Apple provides a number of useful sample projects on the ADC, but for the newcomer it's often like being thrown into the deep end of the pool before you've learned to swim.

My goal in writing this book is to help make the framework approachable and understandable. This book is not intended to be a definitive reference guide covering every aspect of the framework, but instead focuses on the most relevant parts of the framework to lay the foundation that will empower you to be fully comfortable with the concepts, features, and conventions used throughout. It does so by walking you step-by-step through a variety of real-world sample applications ranging from a simple voice memo app to a full-featured video editor similar to iMovie for iOS. It's important to me that you gain a solid grasp of the concepts, and that you also finish the book with a clear understanding of how to use AV Foundation in real-world applications.

Learning AV Foundation is the book I wish I had a few years ago, and I hope it will provide you with the understanding and inspiration to build amazing media applications for iOS and OS X!

—Bob McCune, August 2014

Audience for This Book

The target audience for this book is the experienced Mac or iOS developer who is interested in learning to build digital media applications. It assumes no prior experience with AV Foundation or experience developing media applications, but it does assume you have experience with the frameworks, patterns, and concepts common to developing for Apple's platforms.

Specifically, you should be familiar with the following:

- **C and Objective-C:** The framework is reliant on a number of advanced language and Cocoa features, such as Grand Central Dispatch (GCD), Blocks, and Key-value Observing. You don't need to be a GCD expert, but you should have an understanding of dispatch semantics and the basics of dispatch queues. AV Foundation is an Objective-C framework, but you will commonly work with the framework's supporting C libraries, especially in advanced scenarios, so you should have a working understanding of basic C concepts.
- **Core Animation (optional):** AV Foundation is largely a nonvisual framework, but does have some dependencies on Core Animation for rendering video content. It is helpful, but not required, to have a working knowledge of the Core Animation framework.
- **Drawing/Rendering Frameworks (optional):** Advanced use cases will often integrate with drawing and rendering frameworks, such as Quartz, Core Image, and Open GL or OpenGL ES. The book explains how to integrate with these technologies, but doesn't assume an understanding of how to use these frameworks.

How This Book Is Organized

AV Foundation is a large framework with a broad set of features and capabilities. To help divide the framework into groups of related functionality, the book is organized into three main parts: AV Foundation Essentials, Media Capture, and Media Creation and Editing. The first section covers the foundational aspects of the framework and a number of topics that

are common to most AV Foundation applications. In Media Capture we cover the details of working with the capture APIs to build still and video capture apps. Finally, Media Creation and Editing provides an in-depth look at the capabilities the framework provides to create and edit media.

Here's an overview of what you'll find in the book's chapters:

- **[Chapter 1, “Getting Started with AV Foundation”](#)**—This chapter will help you take your first steps with AV Foundation. It deconstructs the framework to help you gain a better understanding of its features and capabilities. This chapter also provides a high-level overview of the media domain itself and covers topics such as digital sampling and media compression. An understanding of these topics will be helpful throughout the book.
- **[Chapter 2, “Playing and Recording Audio”](#)**—AV Foundation's classes for playing and recording audio are some of its most widely used features. In this chapter we discuss how to use the framework's audio classes, and you'll put them into action building an audio looper and voice memo applications. We also cover how to use audio sessions to help you provide a polished audio user experience to your apps.
- **[Chapter 3, “Working with Assets and Metadata”](#)**—Much of the framework is built around the notion of assets. An asset represents a media resource, such as a QuickTime movie or an MP3 audio file. You learn to use assets and how to use the framework's metadata features by building a metadata editing application.
- **[Chapter 4, “Playing Video”](#)**—Playing video is one of the most essential tasks AV Foundation performs. It's a primary or supporting use case in many media apps. You gain a detailed understanding of how to use the framework's playback features to build a custom video player with full transport controls, subtitle display, and Airplay support.
- **[Chapter 5, “Using AV Kit”](#)**—AV Kit is a new framework introduced in Mac OS X 10.9 and now in iOS 8. It enables you to quickly build AV Foundation video players with user interfaces matching QuickTime Player on OS X and the Videos app on iOS. This can be a great option if you want to build players maintaining fidelity with the native operating system while retaining the full power of working directly with AV Foundation's video APIs covered in [Chapter 4](#).

- **Chapter 6, “Capturing Media”**—This chapter provides an introduction to the framework’s audio and video capture features. You learn to use these features to control the built-in camera hardware available on iOS devices and modern Macs. This is one of the most widely used areas of the framework, and it can help you build powerful, modern camera capture applications.
- **Chapter 7, “Using Advanced Capture Features”**—This chapter covers a variety of advanced capture topics. You learn to use metadata capture to perform barcode scanning and face detection. You learn to use the advanced zooming capabilities provided by the framework. You also learn to enable high frame rate capture, which is great for adding slow motion effects to your videos. We also discuss how to integrate with OpenGL ES to process the video samples captured by the camera, which opens up a world of possibilities.
- **Chapter 8, “Reading and Writing Media”**—AV Foundation provides a lot of high-level functionality, but the framework never hides the lower-level details from you when you need it. In this chapter we discuss the framework’s low-level reading and writing facilities that can enable you to process the media in any way you want. We discuss how to read audio samples from an asset and render them as an audio waveform. We also look at applying real-time video effects using the camera capture APIs.
- **Chapter 9, “Composing and Editing Media”**—In this chapter, we begin our exploration of the framework’s media editing features. This is one of the most powerful features of the framework, and it enables you to create new media by composing and editing media from a variety of sources. You begin building the book’s most advanced application, 15 Seconds, which is a video editor similar to an application such as iMovie for iOS.
- **Chapter 10, “Mixing Audio”**—An important part of building media compositions is learning how to mix multiple audio tracks. You learn how to use mixing techniques such as audio fades and ducking that will help you add polish to your audio presentation.
- **Chapter 11, “Building Video Transitions”**—Video transitions are commonly used to indicate a change in location or storyline, and AV Foundation provides robust support for applying video transitions to

your compositions. In this chapter, you learn to use the framework’s video composition to control the compositing of multiple video tracks in your composition. You’ll put these features into action to add dissolve, push, and wipe transitions to the 15 Seconds app.

- **Chapter 12, “Layering Animated Content”**—This chapter discusses how to add titles, lower thirds, and other animated overlay effects using the Core Animation framework. You’ll see how to use Core Animation to build animation sequences that seamlessly synchronize with your video playback. We also discuss how to incorporate these same effects in your final exported videos.

About the Sample Code

A considerable amount of time was spent developing the book’s sample applications. A big part of learning AV Foundation is gaining an understanding of how it can be used to build real-world applications. To that end, the book includes a large collection of real-world sample projects that you’ll develop throughout the course of this book. These projects can be used as a reference or could even be customized and used as the basis for your own applications. Some of the projects are silly (Hello AVF), some are serious (15 Seconds), but all of them illustrate how to use one or more areas of the framework’s functionality and will be fun for you to build.

AV Foundation is largely the same across both OS X and iOS, so all the sample projects, although written for one platform or the other, are intended to be accessible to developers on both platforms. The sample applications already have their user interfaces and supporting code created, and the code is factored in such a way that you can focus on just the AV Foundation implementation. This makes the sample apps accessible to you regardless of your platform experience, and I think you’ll find it works well from an OO-design standpoint as it helps you develop more reusable, testable code.

The sample projects can be found on my company’s Github site available here: <https://github.com/tapharmonic/Learning-AV-Foundation>

Contacting the Author

You can contact Bob at his website, <http://bobmccune.com>, or you can find him on Twitter (@bobmccune).

Acknowledgments

I would like to thank my mother and father for all their love and support. Whatever I have to give today is because of what they first gave to me. I want to thank my loving wife, Linda, and amazing children, Michael and Kayla. This book would not have been possible without all of their patience and support. Thanks to Trina MacDonald at Pearson Education for her help in making this book possible and patiently guiding me through the process. Thanks to Chris Zahn, Olivia Basegio, Elaine Wiley, and all of the other people at Pearson involved in the development of this title. A special thanks goes to my technical editors, Chris Adamson, Jon Steinmetz, and Ryder Mackay. I greatly appreciated your insight and feedback, and a special shout out goes to Chris for always writing his comments in Comic Sans to make sure I'd make the corrections quickly. Finally, I'd like to thank Apple and its amazing community of Mac and iOS developers. I've been writing software for almost twenty years and have never had as much fun as I am having right now.

About the Author

Bob McCune is an iOS developer and instructor from Minnesota. He started developing for the Mac in 2007 and then switched to iOS when the first iPhone SDK was released in 2008. He is the owner of TapHarmonic, LLC, a small iOS consulting and training company based out of MN. Bob also founded the MN chapter of CocoaHeads in the spring of 2008 and remains the group leader to this day. Bob and his wife, Linda, have two amazing children who are growing up faster than he would like. He is incredibly blessed to have such a loving and supportive family.

I: AV Foundation Essentials

[1 Getting Started with AV Foundation](#)

[2 Playing and Recording Audio](#)

[3 Working with Assets and Metadata](#)

[4 Playing Video](#)

[5 Using AV Kit](#)

1. Getting Started with AV Foundation

Apple has long been a driving force in the world of digital media. In 1991 it introduced QuickTime, which for the first time brought digital audio and video to the masses. The QuickTime architecture would revolutionize digital multimedia for the next two decades, having significant impacts on the education, gaming, and entertainment industries. In 2001 Apple introduced the world to iTunes and the iPod, fundamentally changing the way we listen to music. The iTunes Store, introduced two years later, upended the music industry and has since become the centerpiece of Apple's ever-expanding digital media ecosystem. 2007 brought us the introduction of the iPhone, and a few short years later, the iPad. These events ushered in a whole new era of computing and forever changed the way we create, consume, and share media.

The world of digital media is no longer known only to the technical set. Today, digital media is simple, essential, pervasive, and empowering. Apps such as Instagram make it easy to take beautiful, artistic still images and share them with the world. Video chat applications from Skype to TangoMe bring together friends and family wherever they may be. Streaming video provided by YouTube and Netflix is never more than an LTE or Wi-Fi signal away. And tools like Final Cut Pro X and iMovie for the iPad put the power of video editing in the hands of power users and novices alike.

The digital media revolution is here, but we're just getting started. Learning to use AV Foundation is the key to building the next generation of media applications for Mac OS X and iOS, and this book serves as your guide. It offers an essential overview of the framework, providing you the insight and understanding needed to master the framework. So, let's get started!

What Is AV Foundation?

AV Foundation is Apple's advanced Objective-C framework for working with time-based media on OS X and iOS. It offers a broad and powerful feature set providing you with the tools needed to build modern media applications on Apple's platforms. AV Foundation was built from the beginning with today's hardware and applications in mind. It is designed to be deeply multithreaded. It takes full advantage of multicore hardware and

makes heavy use of blocks and Grand Central Dispatch (GCD) to offload computationally expensive processes to background threads. It automatically provides hardware-accelerated operations ensuring the best possible performance on a wide range of devices. It is designed to be highly power efficient to meet the needs of devices such as the iPhone and iPad. Additionally, it was written to be 64-bit native from the beginning, taking full advantage of 64-bit hardware where available.

Where Does AV Foundation Fit?

One of the first steps to learning AV Foundation is to get a clear understanding of where it fits within Apple's overall media landscape. Mac OS X and iOS provide developers with a number of high-level and low-level frameworks for working with timed media. [Figure 1.1](#) shows how AV Foundation fits into the overall picture.

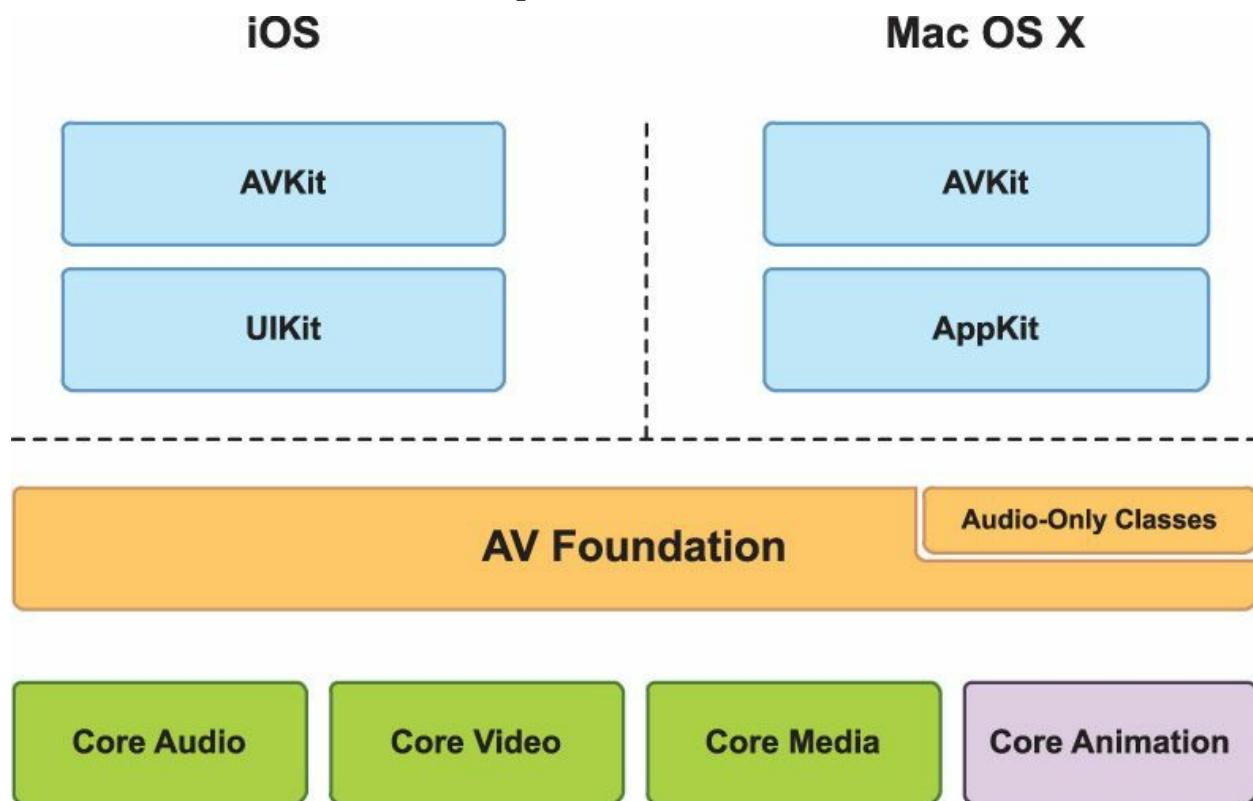


Figure 1.1 Mac OS X and iOS media environment

Both platforms offer a number of high-level solutions for working with media. On iOS, the UIKit framework makes it easy to incorporate basic still image and video capture into your applications. Both Mac OS X and iOS can make use of the HTML5 <audio> and <video> tags inside either a WebView

or UIWebView to play audio and video content. Both platforms additionally provide the AVKit framework, which simplifies building modern video playback applications. All these solutions are convenient and easy to use and should be considered when adding media functionality into your applications. However, although these solutions are convenient, they often lack the flexibility and control needed by more advanced applications.

At the other end of the spectrum are several lower-level frameworks that provide supporting functionality used by all the higher-level solutions. Most of these are low-level, procedural C-based frameworks that are incredibly powerful and performant, but are complex to learn and use and require a strong understanding of how media is processed at a hardware level. Let's look at some of the key supporting frameworks and the functionality each provides.

▪ **Core Audio**

Core Audio handles all audio processing on OS X and iOS. Core Audio is a suite of frameworks providing interfaces for the recording, playback, and processing of audio and MIDI content. Core Audio provides both higher-level interfaces, such as those provided by the Audio Queue Services framework, which can be used for basic audio playback and recording needs. It also provides very low-level interfaces, specifically Audio Units, which provide complete control over the audio signal and enable you to build sophisticated audio processing features like those used by tools such as Apple's Logic Pro X and Avid's Pro Tools. For an excellent overview of this topic, I highly recommend reading *Learning Core Audio* by Chris Adamson and Kevin Avila (2012, Boston: Addison-Wesley).

▪ **Core Video**

Core Video provides a pipeline model for digital video on OS X and iOS. It provides image buffer and buffer-pool support to its counterpart, Core Media, providing it an interface for accessing the individual frames in a digital video. It simplifies working with this data by translating between pixel formats and managing video synchronization concerns.

▪ **Core Media**

Core Media is part of the low-level media pipeline used by AV

Foundation. It provides the low-level data types and interfaces needed for working with audio samples and video frames. Core Media additionally provides the timing model used by AV Foundation based around the `CMTIME` data type. `CMTIME`, and its associated data types, are used when working with time-based operations in AV Foundation.

▪ **Core Animation**

Core Animation is the compositing and animation framework provided on OS X and iOS. The behavior it provides is essential to the beautiful, fluid animations seen on Apple's platforms. It offers a simple, declarative programming model providing an Objective-C wrapper over functionality enabled by OpenGL and OpenGL ES. Using Core Animation, AV Foundation provides hardware-accelerated rendering of video content in both playback and video capture scenarios. AV Foundation additionally makes use of Core Animation, enabling you to add animated titling and image effects in video editing and playback scenarios.

Sitting between the high-level and low-level frameworks is AV Foundation. The positioning of AV Foundation within the overall media landscape is significant. It offers much of the power and performance of the lower-level frameworks, but in a much simpler Objective-C interface. It can work seamlessly with higher-level frameworks, such as Media Player and the Assets Library, making use of the services they provide, and at the same time it can interact directly with Core Media and Core Audio when more advanced needs arise. Additionally, because AV Foundation sits below the UIKit and AppKit layers, it also means you have a single media framework to use on both platforms. There is only one framework to learn, providing you the opportunity to port not only your code, but also your knowledge and experience to either platform.

Decomposing AV Foundation

One of the biggest early challenges in learning to use AV Foundation is making sense of the large number of classes the framework provides. The framework contains more than 100 classes, a large collection of protocols, and a variety of functions and constants you'll use as well. This can certainly seem a bit overwhelming the first time it is encountered, but when you decompose the framework into its functional units it becomes much more

understandable. Let's look at the key areas of functionality it provides.

Audio Playback and Recording

If you look back at [Figure 1.1](#), you'll see a small box in the upper-right corner of the AV Foundation box labeled Audio-Only Classes. Some of the earliest functionality provided by AV Foundation relates to audio.

`AVAudioPlayer` and `AVAudioRecorder` provide easy ways of incorporating audio playback and recording into your applications. These aren't the only ways of playing and recording audio in AV Foundation, but they are the easiest to learn and provide some powerful features.

Media Inspection

AV Foundation provides the capability to inspect the media you are using. You can inspect media assets to determine their suitability for a particular task, such as whether they can be used for playback or if they can be edited or exported. You can retrieve technical attributes about the media, such as its duration, its creation date, or its preferred playback volume. Additionally, the framework provides powerful metadata support based around the `AVMetadataItem` class. This enables you to read and write descriptive metadata about the media, such as album and artist information.

Video Playback

One of the more common uses of AV Foundation is to provide video playback. This is often a primary or secondary use case in many media applications. The framework enables you to play video assets from either a local file or a remote stream, and control the playback and display of the video content. The central classes in this area are the `AVPlayer` and `AVPlayerItem` classes that enable you to control the playback of an asset, as well as incorporate more advanced features, such as subtitles and chapter information. Or you can access alternate audio and video tracks.

Media Capture

These days, almost all Macs and all iOS devices include built-in cameras. These are high quality devices that can be used for capturing both still and video images. AV Foundation provides a rich set of APIs, giving you fine-grained control of the capabilities of these devices. The central class in

capture scenarios is `AVCaptureSession`, which is the central hub of activity for routing camera device output to movie and image files as well as media streams. This has always been a robust area of functionality within AV Foundation and has been significantly enhanced again in the most recent release of the framework.

Media Editing

AV Foundation also provides very strong support for media composition and editing. It enables you to create applications that can compose multiple tracks of audio and video together, trim and edit individual media clips, modify audio parameters over time, and add animated title and transition effects. Tools such as Final Cut Pro X and iMovie for the Mac and iPad are prime examples of the kind of applications that can be built using this functionality.

Media Processing

Although much can be accomplished in AV Foundation without getting too deeply into the bits and bytes of the media, at times you need to get access to this level of detail. Fortunately, when you need to perform more advanced media processing, you can do so using the `AVAssetReader` and `AVAssetWriter` classes. These classes provide direct access to the video frames and audio samples, so you can perform any kind of advanced processing you require.

Understanding Digital Media

These days it's easy to take digital media for granted. We buy songs and albums from iTunes, stream movies and TV shows from Netflix and Hulu, and share digital photos by email, text, and on the Web. Using digital media has become second nature for most of us, but have you ever given much thought to how that media became digital in the first place? We clearly live in a digital age, but we still inhabit an analog world. Every sight that we see and every sound that we hear is delivered to us as an analog signal. The inner structures of our eyes and ears convert these signals into electrical impulses that our brains perceive as sight and sound. Signals in the real world are *continuous*, constantly varying in frequency and intensity, whereas signals in the digital world are *discrete*, having a state of either 1 or 0. In order to translate an analog signal into a form that we can store and transmit digitally,

we use an analog-to-digital conversion process called *sampling*.

Digital Media Sampling

There are two primary types of sampling used when digitizing media. The first is called **temporal sampling**, which enables us to capture variations in a signal over time. For instance, when you record a voice memo on your iPhone, the continuous variations in the pitch and volume of your voice are being captured over the duration of your recording. The second type of sampling is called **spatial sampling** and is used when digitizing photographs or other visual media. Spatial sampling involves capturing the luminance (light) and chrominance (color) in an image at some degree of resolution in order to create the resulting digital image's pixel data. When digitizing video, both forms of sampling are used because a video signal varies both spatially and temporally.

Fortunately, you don't need to have a deep understanding of the complex digital signal processing involved in these sampling processes, because it is handled by the hardware components that perform the analog-to-digital conversion. However, failing to have a basic understanding of these processes and the storage formats of the digital media they produce will limit your ability to utilize some of AV Foundation's more advanced and interesting capabilities. To get a general understanding of the sampling process, let's take a look at the steps involved in sampling audio.

Understanding Audio Sampling

When you hear the sound of someone's voice, the honking of a horn, or the strum of a guitar, what you are really hearing are vibrations transmitted through sound waves over some medium. For instance, when you strum a G chord on a guitar, as the guitar pick strikes the strings, it causes each string to vibrate at a certain frequency and amplitude. The speed or frequency at which the string vibrates back and forth determines its pitch, with low notes producing low, slow-modulating frequencies and high notes producing high, fast-modulating frequencies. The amplitude measures the relative magnitude of the frequency, which roughly correlates to the volume you hear. On a stringed instrument such as a guitar, you can actually *see* both the frequency and amplitude attributes of the signal when you pluck the string. This vibration causes the surrounding air molecules to move, which in turn push

against their neighboring molecules, which push against their neighbors, and so on, continuously transmitting the energy from the initial vibration outward in all directions. As these waves reach your ear, they cause your eardrum to vibrate at the same frequency and amplitude. These vibrations are transmitted to the cochlea in your inner ear, where they are converted into electrical impulses sent to your brain, causing you to think, “I’m hearing a G chord!”

When we record a voice, an acoustic instrument such as a piano or a guitar, or capture other environmental sounds, we use a microphone. A microphone is a *transducer* that translates mechanical energy (a sound wave) into electrical energy (voltage). A variety of different microphone types are in use, but I’ll discuss this in terms of one called a *dynamic* microphone. [Figure 1.2](#) shows a high-level view of the internals of a dynamic microphone.

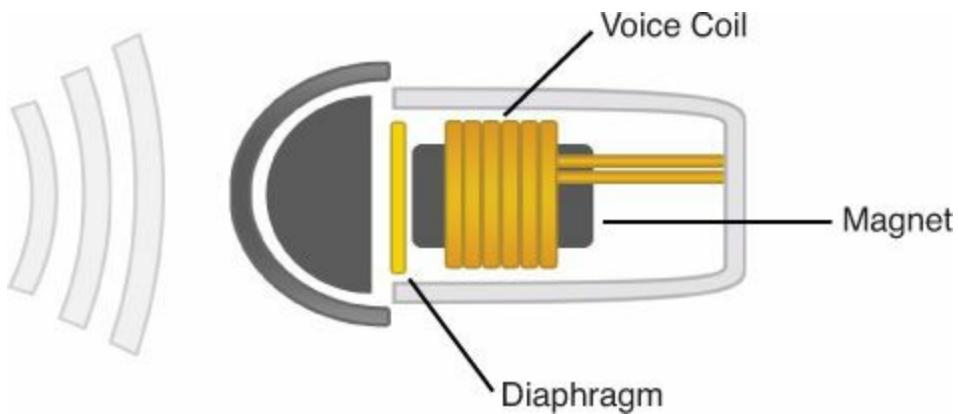


Figure 1.2 Internal view of a dynamic microphone

Contained inside the head case, which is the part you speak into, is a thin membrane called a diaphragm. The diaphragm is connected to a coil of wire wrapped around a magnet. When you speak into the microphone, the diaphragm vibrates in relationship to the sound waves it senses. This in turn vibrates the coil of wire, causing a current to be generated relative to the frequency and amplitude of the input signal. Using an oscilloscope, we can see the oscillations of this current, as shown in [Figure 1.3](#).

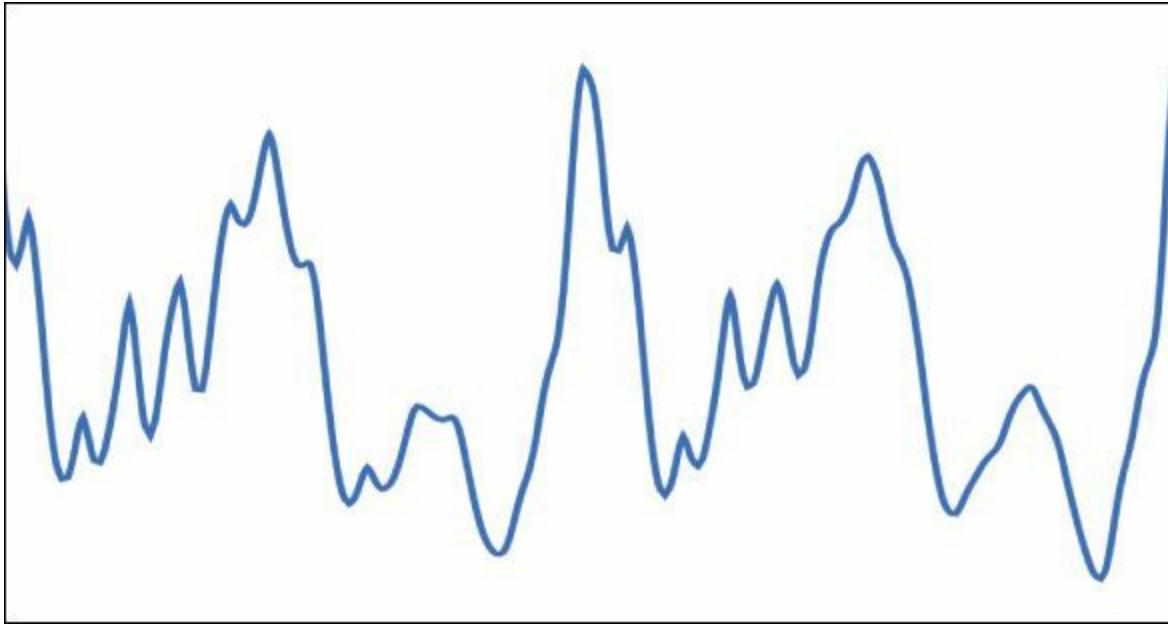


Figure 1.3 Audio signal voltage

Returning to the topic of sampling, how do we convert this continuous signal into its discrete form? Let's drill in a bit further into the essential element in an audio signal. Using a tone generator, I created two different tones producing the sine waves shown in [Figure 1.4](#).

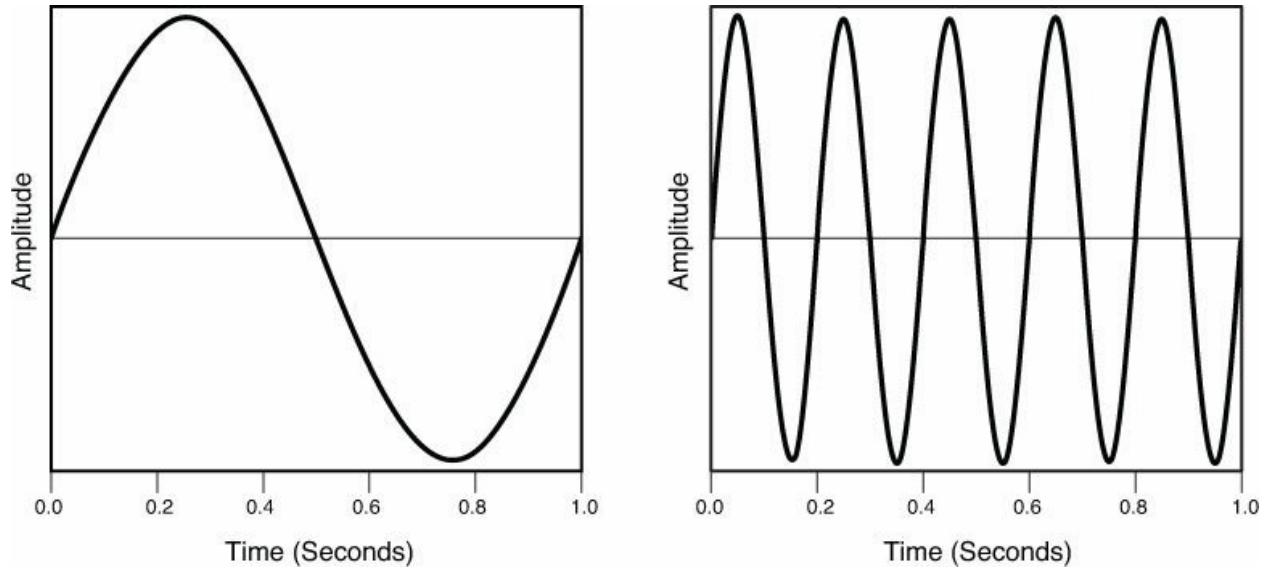


Figure 1.4 Sine waves at 1Hz (left) and 5Hz (right)

We're interested in two aspects of this signal. The first is the *amplitude*, which indicates the magnitude of the voltage or relative strength of the signal. This can be represented on a variety of scales, but is commonly normalized to a range of $-1.0f$ to $1.0f$. The other interesting aspect of this signal is its

frequency. The frequency of the signal is measured in hertz (Hz), which indicates how many complete cycles occur in the period of one second. The image on the left in [Figure 1.4](#) shows an audio signal cycling at 1Hz and the one on the right shows a 5Hz signal. Humans have an audible frequency range of 20Hz–20kHz (20,000 Hz), so both signals would be inaudible, but they make for easier illustration.

Note

Although human hearing has an audible frequency range of 20Hz to 20kHz, that range really represents theoretical boundaries. Few people can hear frequencies in the outer bounds of that range, because hearing declines if you're exposed to loud environments, and it declines rapidly as you age. If you've ever been to a rock concert, I can assure you that the upper part of that range is gone.

To provide some frame of reference for the sound of various frequencies, the lowest key on a piano, A0, produces a frequency of 27.5Hz and C8, the highest key, produces a frequency of approximately 4.1kHz.

Digitizing audio involves a method of encoding called *linear pulse-code modulation*, more commonly referred to as Linear PCM or LPCM. This process samples or measures the amplitude of an audio signal at a fixed, periodic rate called the *sampling rate*. [Figure 1.5](#) shows taking seven samples of this signal over the period of 1 second and the resulting digital representation of the signal.

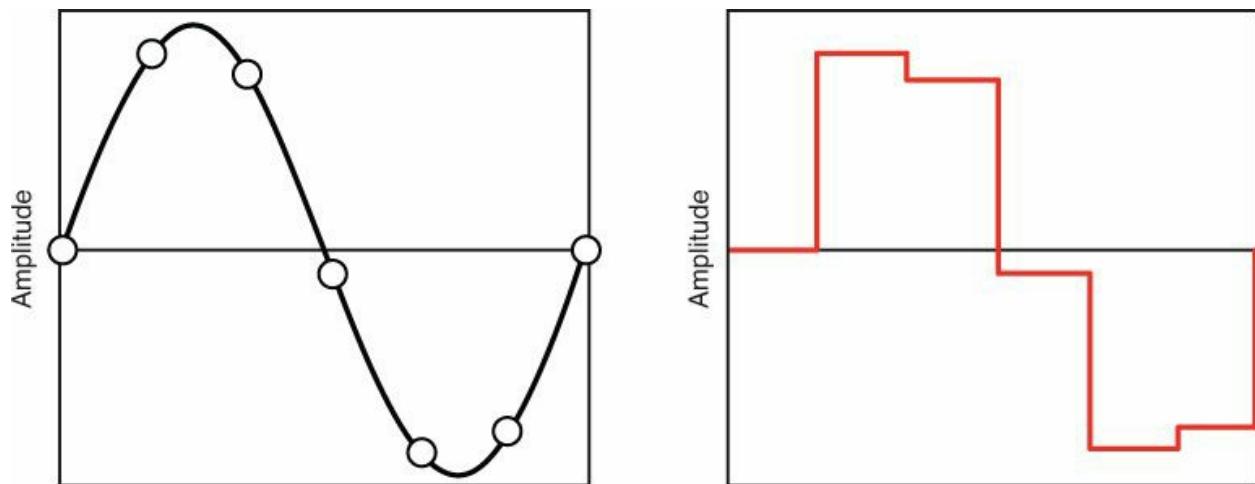


Figure 1.5 Low sampling rate

Clearly, at a low sampling rate the digital version of this signal bears little resemblance to the original. Playing this digital audio would result in little more than clicks and pops. The problem with the sampling shown in [Figure 1.5](#) is that it isn't sampling frequently enough to accurately capture the signal. Let's try this again in [Figure 1.6](#), but this time we'll increase the sampling rate.

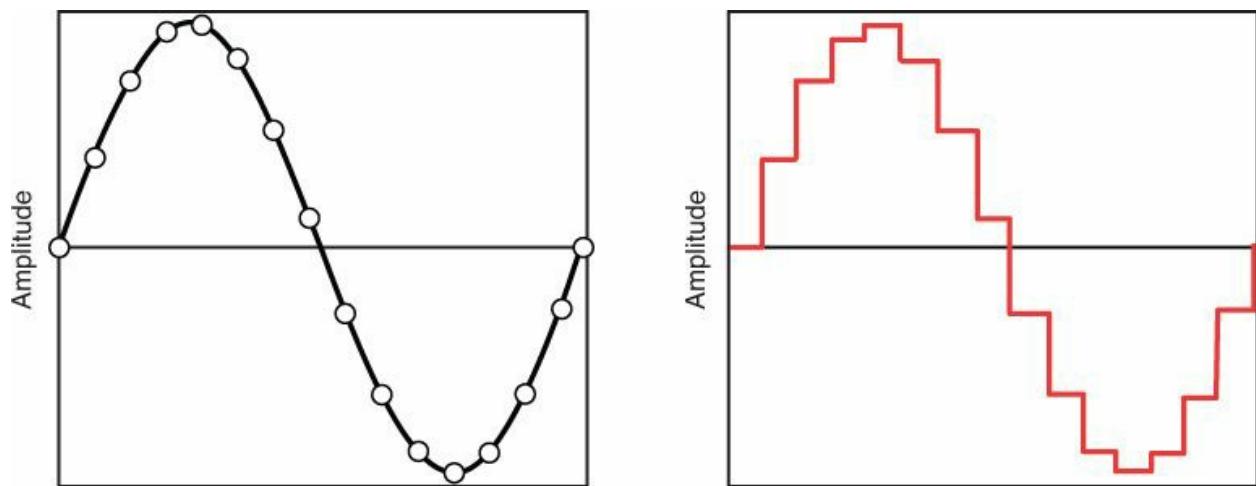


Figure 1.6 Higher sampling rate

This is certainly an improvement, but still not a very accurate representation of the signal. However, what you can surmise from this example is if you continue to increase the frequency of the sample rate, we should be able to produce a digital representation that fairly accurately mirrors the original source. Given the limitations of hardware, we may not be able to produce an exact replica, but is there a sample rate that can produce a digital representation that is good enough? The answer is yes, and it's called the *Nyquist rate*. Harry Nyquist was an engineer working for Bell Labs in the 1930s who discovered that to accurately capture a particular frequency, you need to sample at a rate of at least twice the rate of the highest frequency. For instance, if the highest frequency in the audio material you wanted to capture is 10kHz, you need a sample rate of at least 20kHz to provide an accurate digital representation. CD-quality audio uses a sampling rate of 44.1kHz, which means that it can capture a maximum frequency of 22.05kHz, which is just above 20kHz upper bound of human hearing. A sampling rate of 44.1kHz may not capture the complete frequency range contained in the source material, meaning your dog may be upset by the recording because it

doesn't capture the nuances of the Abbey Road sessions, but for us human beings, it sounds pristine.

In addition to the sampling rate, another important aspect of digital audio sampling is how accurately we can capture each audio sample. The amplitude is measured on a *linear* scale, hence the term Linear PCM. The number of bits used to store the sample value defines the number of discrete steps available on this linear scale and is referred to as the audio's *bit depth*.

Assigning too few bits results in considerable rounding or quantizing of each sample, leading to noise and distortion in the digital audio signal. Using a bit depth of 8 would provide 256 discrete levels of quantization. This may be sufficient for some audio material, but it isn't high enough for most audio content. CD-quality audio has a bit depth of 16, resulting in 65,536 discrete levels, and in professional audio recording environments bit depths of 24 or higher are used.

When we digitize a signal, we are left with its raw, uncompressed digital representation. This is the media's purest digital form, but it requires significant storage space. For instance, a 44.1kHz, 16-bit LPCM audio file takes about 10MB per stereo minute. To digitize a 12-song album with the average song length of 5 minutes would take approximately 600MB of storage. Even with the vast amounts of storage and bandwidth we have today, that is still pretty large. We can see that uncompressed digital audio requires significant amounts of storage, but what about uncompressed video? Let's take a look at the elements of a digital video to see if we can determine the amount of storage space it requires.

Video is composed of a sequence of images called *frames*. Each frame captures a scene for a point in time within the video's timeline. To create the illusion of motion, we need to see a certain number of frames played in fast succession. The number of frames displayed in one second is called video's *frame rate* and is measured in frames per second (FPS). Some of the most common frame rates are 24FPS, 25FPS, and 30FPS.

To understand the storage requirements for uncompressed video content, we first need to determine how big each individual frame would be. A variety of common video sizes exist, but these days they usually have an *aspect ratio* of 16:9, meaning there are 16 horizontal pixels for every 9 vertical pixels. The two most common sizes of this aspect ratio are 1280×720 and 1920×1080 . What about the pixels themselves? If we were to represent each pixel in the

RGB color space using 8 bits, that means we'd have 8 bits for red, 8 bits for green, and 8 bits for blue, or 24 bits. With all the inputs gathered, let's perform some calculations. [Table 1.1](#) shows the storage requirements for uncompressed video at 30FPS at the two most common resolutions.

Color	Resolution	Frame Rate	MB/sec	GB/hour
24-bit	1280 × 720	30FPS	79MB/sec	278GB/hr
24-bit	1920 × 1080	30FPS	178MB/sec	625GB/hr

Table 1.1 Uncompressed Video Storage Requirements

Houston, we have a problem. Clearly, as a storage and transmission format, this would be untenable. A decade from now these sizes may seem trivial, but today this isn't feasible for most uses. Because this isn't a reasonable way to store and transfer video in most cases, we need to find way to reduce this size. This brings us to the topic of compression.

Digital Media Compression

To reduce the size of digital media we need to use compression. Virtually all the media we consume is compressed to various degrees. Whether it's video on TV, a Blu-ray disc, streamed over the web, or purchased from the iTunes Store, we're dealing with compressed formats. Compressing digital media can result in greatly reduced file sizes, but often with little or no perceivable degradation in quality.

Chroma Subsampling

Video data is typically encoded using a color model called $Y'C_bC_r$,—which is commonly referred to as YUV. The term *YUV* is technically incorrect, but YUV probably rolls off the tongue better than Y-Prime-C-B-C-R. Most software developers are more familiar with the RGB color model, where every pixel is composed of some value of red, green, and blue. $Y'C_bC_r$, or YUV, instead separates a pixel's *luma* channel Y (brightness) from its chroma (color) channels UV. [Figure 1.7](#) illustrates the effect of separating an image's luma and chroma channels.



Figure 1.7 Original image on the left. Luma (Y) in the center. Chroma (UV) on the right.

You can see that all the detail of the image is preserved in the luma channel, leaving us with a grayscale image, whereas in the combined chroma channels almost all the detail is lost. Because our eyes are far more sensitive to brightness than they are to color, clever engineers over the years realized we can reduce the amount of color information stored for each pixel while still preserving the quality of the image. The process used to reduce the color data is called *chroma subsampling*.

Whenever you see camera specifications or other video hardware or software referring to numbers such as 4:4:4, 4:2:2, or 4:2:0, these values refer to the chroma subsampling it uses. These values express a ratio of luminance to chrominance in the form J:a:b where

- J: is the number of pixels contained within some reference block (usually 4).
- a: is number of chrominance pixels that are stored for every J pixels in the first row.
- b: is the number of additional pixels that are stored for every J pixels in the second row.

To preserve the quality of the image, every pixel needs to have its own luma value, but it does not need to have its own chroma value. [Figure 1.8](#) shows the common subsampling ratios and the effects of each.

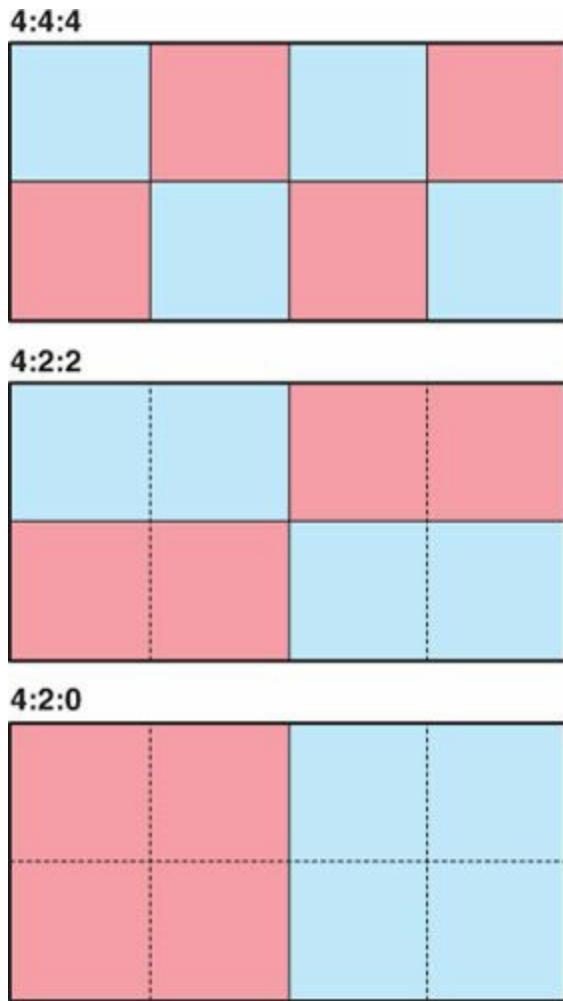


Figure 1.8 Common chroma subsampling ratios

In all forms, full luminance is preserved across all pixels, and in 4:4:4 full color information is preserved as well. In 4:2:2, color information is averaged across every two pixels horizontally, resulting in a 2:1 luma-to-chroma ratio. In 4:2:0, color information is averaged both horizontally and vertically, resulting in a 4:1 luma-to-chroma ratio.

Chroma subsampling typically happens at the point of acquisition. Some professional cameras capture at 4:4:4, but more commonly they do so at 4:2:2. Consumer-oriented cameras, such as the one found on the iPhone, capture at 4:2:0. A high-quality image can be captured even at significant levels of subsampling, as is evidenced by the quality of video that can be shot on the iPhone. The loss of color becomes more problematic when performing chroma keying or color correction in the post-production process. As the chroma information is averaged across multiple pixels, noise and other artifacts can enter into the image.

Codec Compression

Most audio and video is compressed with the use of a codec, which is short for encoder/decoder. A codec is used to encode audio or video data using advanced compression algorithms to greatly reduce the size needed to store or deliver digital media. The codec is also used to decode the media from its compressed state into one suitable for playback or editing.

Codecs can be either *lossless* or *lossy*. A lossless codec compresses the media in a way that it can be perfectly reconstructed upon decompression, making it ideal for editing and production uses, as well as for archiving purposes. We use this type of compression frequently when using utilities like zip or gzip. A lossy codec, as the name suggests, loses data as part of the compression process. Codecs employing this form of compression use advanced algorithms based on human perception. For instance, although we can theoretically hear frequencies between 20Hz and 20kHz, we are particularly sensitive to frequencies between 1kHz and 5kHz. Our sensitivity to the frequencies begins to taper off as we get above or below this range. Using this knowledge, an audio codec can employ filtering techniques to reduce or eliminate certain frequencies in an audio file. This is just one example of the many approaches used, but the goal of lossy codecs is to use psycho-acoustic or psycho-visual models to reduce redundancies in the media in a way that will result in little or no *perceivable* degradation in quality.

Let's look at the codec support provided by AV Foundation.

Video Codecs

AV Foundation supports a fairly limited set of codecs. It supports only those that Apple considers to be the most relevant for today's media. When it comes to video, that primarily boils down to H.264 and Apple ProRes. Let's begin by looking at H.264 video.

H.264

When it comes to encoding your video for delivery, I'll paraphrase Henry Ford by saying AV Foundation supports any video codec you want as long as it's H.264. Fortunately, the industry has coalesced around this codec as well. It is widely used in consumer video cameras and is the dominant format used for video streaming on the Web. All the video downloaded from the iTunes Store is encoded using this codec as well. The H.264 specification is part of

the larger MPEG–4 part 14 specification defined by the Motion Picture Experts Group (MPEG). H.264 builds on the earlier MPEG–1 and MPEG–2 standards, but provides greatly improved image quality at lower bit rates, making it ideal for streaming and for use on mobile devices and video cameras.

H.264, along with other forms on MPEG compression, reduces the size of video content in two ways:

- **Spatially:** This compresses the individual video frames and is referred to as *intraframe compression*.
- **Temporally:** Compresses redundancies across groups of video frames. This is called *interframe compression*.

Intraframe compression works by eliminating redundancies in color and texture contained within the individual video frames, thereby reducing their size but with minimal loss in picture quality. This form of compression works similarly to that of JPEG compression. It too is a lossy compression algorithm, but can be used to produce very high-quality photographic images at a fraction of the size of the original image. The frames created through this process are referred to as *I-frames*.

With interframe compression, frames are grouped together into a **Group of Pictures (GOP)**. Within this GOP certain temporal redundancies exist that can be eliminated. If you think about a typical scene in video, there are certain elements in motion, such as a car driving by or a person walking down the street, but the background environment is often fixed. The fixed background represents a temporal redundancy that could be eliminated through compression.

There are three types of frames that are stored within a GOP, as shown in [Figure 1.9](#).

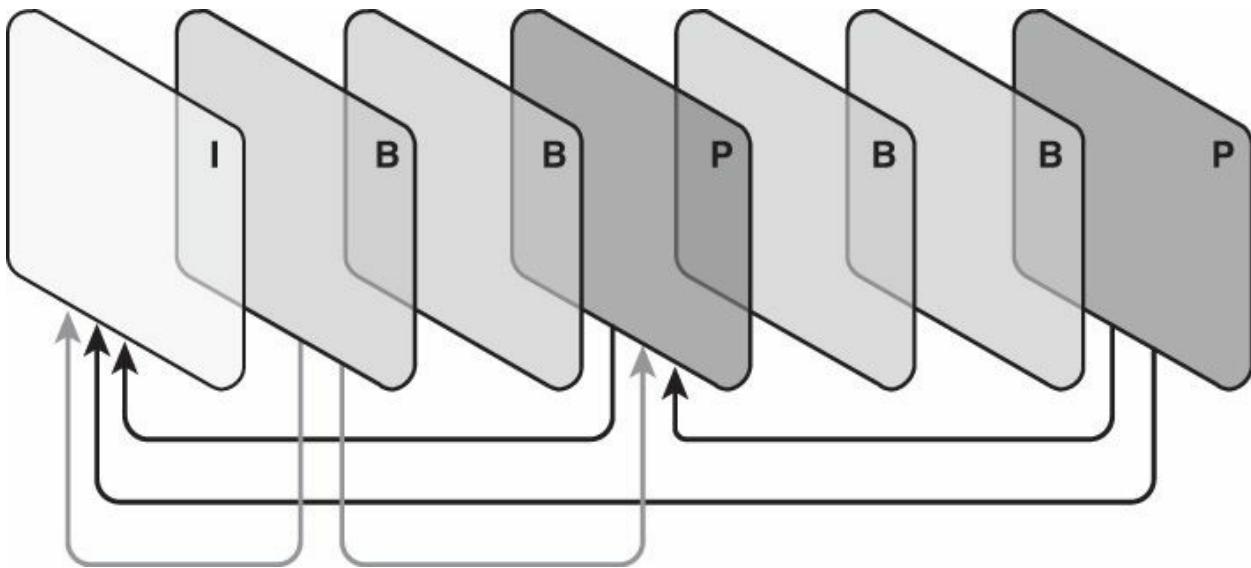


Figure 1.9 Group of Pictures

- **I-frames:** These are the standalone or *key* frames and contain all the data needed to create the complete image. Every GOP has exactly one I-frame. Because it is a standalone frame, it is the largest in size but is fastest to decompress.
- **P-frames:** P-frames, or *predicted frames*, are encoded from a “predicted” picture based on the closest I-frame or P-frame. P-frames can reference the data in the closest preceding P-frame or the group’s I-frame. You’ll often see these referred to as *reference frames*, as their neighboring P-frames and B-frames can refer to them.
- **B-frames:** B-frames, or *bidirectional frames*, are encoded based on frame information that comes before and after them. They require little space, but take longer to decompress because they are reliant on their surrounding frames.

H.264 additionally supports encoding profiles, which determine the algorithms employed during the encoding process. There are three top-level profiles defined:

- **Baseline:** This profile is commonly used when encoding media for mobile devices. It provides the least efficient compression, thereby resulting in larger file sizes, but is also the least computationally intensive because it doesn’t support B-frames. If you’re targeting older iOS devices, such as the iPhone 3GS, you should use the baseline profile.

- **Main:** This profile is more computationally intensive than baseline, because a greater number of its available algorithms are used, but it results in higher compression ratios.
- **High:** The high profile will result in the highest quality compression being used, but is the most intensive of the three because the full arsenal of encoding techniques and algorithms are used.

Apple ProRes

AV Foundation supports two flavors of the Apple ProRes codec. Apple ProRes is considered an intermediate or mezzanine codec, because it's intended for professional editing and production workflows. Apple ProRes codecs are frame-independent, meaning only I-frames are used, making it more suitable for editing. They additionally use variable bit rate encoding that varies the number of bits used to encode each frame based on the complexity of the scene.

ProRes is a lossy codec, but of the highest quality. Apple ProRes 422 uses 4:2:2 chroma subsampling and a 10-bit sample depth. Apple ProRes 4444 uses 4:4:4 chroma subsampling, with the final 4 indicating it supports a lossless alpha channel and up to a 12-bit sample depth.

The ProRes codecs are available only on OS X. If you're developing only for iOS, H.264 is the only game in town. Apple does, however, provide one variation to typical H.264 encoding that can be used when capturing for editing purposes—called iFrame. This is an *I-frame-only* variant producing H.264 video more suitable for editing environments. This format is supported within AV Foundation and is additionally supported by a variety of camera manufacturers, such as Canon, Panasonic, and Nikon.

Note

In addition to H.264 and Apple ProRes, AV Foundation supports a number of camera device codecs, such as MPEG–1, MPEG–2, MPEG–4, H.263, and DV, enabling you to import content from a variety of video cameras.

Audio Codecs

AV Foundation supports all the audio codecs supported by the Core Audio

framework, meaning it has broad support for a variety of formats. However, when you're not using linear PCM audio, the one you will most frequently use is AAC.

AAC

Advanced Audio Coding (AAC) is the audio counterpart to H.264 and is the dominant format used for audio streaming and downloads. It greatly improves upon MP3, providing higher sound quality at lower bit rates, which makes it ideal for distribution on the Web. Additionally, AAC doesn't have the licensing and patent restrictions that have long plagued MP3.

Note

AV Foundation and Core Audio provide support for decoding MP3 data, but they do not provide the capability of encoding it.

Container Formats

If you're like most people, you're likely to find a variety of media files on your computer. You'll find files with extensions such as .mov, .m4v, .mpg, and .m4a. Although we commonly refer to these types as file formats, the correct definition is they are *container* formats.

A container format is considered a metafile format. From a high level you can think of a container format as a directory containing one or more types of media along with metadata describing its contents. A QuickTime file, for instance, can contain a variety of media types, including video, audio, subtitles, and chapter information, and contains metadata describing the details of each piece of media it holds.

Each format has a specification that determines the structure of the file. The structure defines not only the technical aspects of the media it contains, such as the media's duration, encoding, and timing information, but also commonly defines descriptive metadata, such as a movie's title or a song's artist information. This metadata can be presented in tools such as iTunes or the iOS Music app, and AV Foundation provides the classes to read and write this type of data in your applications as well.

You'll use two primary container formats when working with AV Foundation:

- **QuickTime:** QuickTime is Apple's proprietary format defined as part of the larger QuickTime architecture. This is an extremely robust and highly specified format that is widely used in both professional and consumer settings. Apple describes this format in great detail in a QuickTime File Format Specification document that you can find on the Apple Developer Connection site. I recommend that all AV Foundation developers read at least the introductory sections of this document because it provides valuable insight that will benefit you when developing media applications.
- **MPEG-4:** The MPEG-4 Part 14 specification defines the MPEG-4 (MP4) container format. This is an industry standard format derived directly from the QuickTime specification, so the two are very similar in structure and capabilities. The official file extension defined for an MP4 container is .mp4 but a variety of variant extensions are in use, particularly within Apple's ecosystem. These variant file extensions still use the same basic MP4 container format, but are often used to distinguish the particular media type, as is the case with an m4a audio file, or can additionally indicate the use of extensions to the base MP4 container, as is the case with m4v video files.

Hello AV Foundation

Now that you have a high-level understanding of AV Foundation and some deeper insight into the details of digital media, let's wrap up this chapter by having a little fun.

Mac OS X has long had the NSSpeechSynthesizer class, making it easy to add text-to-speech features in Cocoa applications. You can add similar functionality to your iOS apps using AV Foundation's AVSpeechSynthesizer class. This class is used to speak one or more *utterances*, which are instances of a class called AVSpeechUtterance. If you wanted to speak the phrase "Hello World!" you could do so as follows:

[Click here to view code image](#)

```
AVSpeechSynthesizer *synthesizer = [[AVSpeechSynthesizer alloc]
init];
AVSpeechUtterance *utterance =
[[AVSpeechUtterance alloc] initWithString:@"Hello World!"];
[synthesizer speakUtterance:utterance];
```

If you ran this code, you would hear the phrase “Hello World!” being spoken in the default voice for your locale. Let’s put this functionality into action by building a simple app that will carry on a conversation with AV Foundation. All the projects you’ll build throughout this book have a “starter” and “final” version in the book’s sample code repository. The final version is the completed project and is ready to build and run. The starter version has the user interface and supporting classes completed and contains stubbed versions of the classes you’ll be developing. Additionally, most of the sample projects have the code factored in a way to isolate the AV Foundation code from the rest of the application. This will make it easy for us to stay focused on AV Foundation without getting bogged down in the user interface details; it also makes the sample apps accessible to you whether your primary experience is in OS X or iOS.

In the book’s sample code repository, you’ll find a starter project in the Chapter 1 directory called **HelloAVF_Starter**. [Figure 1.10](#) shows this app in action.



Figure 1.10 Hello AV Foundation!

In the project you'll find a class called `THSpeechController`. This is the class in which you'll develop the application's text-to-speech functionality. [Listing 1.1](#) shows the interface for this class.

Listing 1.1 `THSpeechController.h`

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>

@interface THSpeechController : NSObject

@property (strong, nonatomic, readonly) AVSpeechSynthesizer
*synthesizer;

+ (instancetype)speechController;
```

```
- (void)beginConversation;  
@end
```

This class has a simple interface with just a couple points to note. The header begins with an import of `<AVFoundation/AVFoundation.h>`, which is the umbrella header for the framework. This will be a common fixture in all the code you write throughout the course of this book. The key method in this class is `beginConversation`, which will kick off the text-to-speech functionality you'll be building in a minute and put the app into action. Let's switch over to the class implementation (see [Listing 1.2](#)).

Listing 1.2 **THSpeechController.m**

[Click here to view code image](#)

```
#import "THSpeechController.h"  
#import <AVFoundation/AVFoundation.h>  
  
@interface THSpeechController()  
@property (strong, nonatomic) AVSpeechSynthesizer  
*synthesizer; // 1  
@property (strong, nonatomic) NSArray *voices;  
@property (strong, nonatomic) NSArray *speechStrings;  
@end  
  
@implementation THSpeechController  
  
+ (instancetype)speechController {  
    return [[self alloc] init];  
}  
  
- (id)init {  
    self = [super init];  
    if (self) {  
        _synthesizer = [[AVSpeechSynthesizer alloc]  
init]; // 2  
  
        _voices = @[[AVSpeechSynthesisVoice  
voiceWithLanguage:@"en-US"], // 3  
                    [AVSpeechSynthesisVoice  
voiceWithLanguage:@"en-GB"]];  
  
        _speechStrings = [self buildSpeechStrings];  
    }  
    return self;  
}
```

```

- (NSArray *)buildSpeechStrings
{
    return @[@"Hello AV Foundation. How are you?", // 4
             @"I'm well! Thanks for asking.",
             @"Are you excited about the book?",
             @"Very! I have always felt so misunderstood",
             @"What's your favorite feature?",
             @"Oh, they're all my babies. I couldn't possibly
choose.",
             @"It was great to speak with you!",
             @"The pleasure was all mine! Have fun!"];
}

- (void)beginConversation {

}

@end

```

- 1.** Define the class's required properties in the class extension, redefining the `synthesizer` property that was defined in the header so that it's read/write. Additionally, define properties for the voices and speech strings that will be used in the conversation.
- 2.** Create a new instance of `AVSpeechSynthesizer`. This is the object performing the text-to-speech conversion. It acts as a queue for one or more instances of `AVSpeechUtterance` and provides you with the interface to control and monitor the progress of the ongoing speech.
- 3.** Create an `NSArray` containing two instances of `AVSpeechSynthesisVoice`. Voice support is currently very limited. You don't have the ability to specify *named* voices like you can on the Mac. Instead, each language/locale has one predefined voice. In this case, speaker #1 will use the U.S. English voice and speaker #2 will use the British English voice. You can get a complete listing of supported voices by calling the `speechVoices` class method on `AVSpeechSynthesisVoice`.
- 4.** Create an array of strings defining the back and forth of the contrived conversation.

With the basic set up of the class complete, let's move on and discuss the implementation of the `beginConversation` method, as shown in [Listing](#)

1.3.

Listing 1.3 Implementing the `beginConversation` Method

[Click here to view code image](#)

```
- (void)beginConversation {
    for (NSUInteger i = 0; i < self.speechStrings.count; i++) {
        AVSpeechUtterance *utterance
        =
        [[AVSpeechUtterance alloc]
initWithString:self.speechStrings[i]];
        utterance.voice = self.voices[i %
2];
        utterance.rate =
0.4f;
        utterance.pitchMultiplier =
0.8f;
        utterance.postUtteranceDelay =
0.1f;
        [self.synthesizer
speakUtterance:utterance];
    }
}
```

1. Loop through the collection of speech strings, and for each you'll create a new instance of `AVSpeechUtterance`, passing the string to its `initWithString:` initializer.
2. Toggle back and forth between the two voices you defined previously. Even iterations will speak in the U.S. voice and odd iterations will speak in the British voice.
3. Specify the rate at which this utterance will be spoken. I'm setting this to a value of `0.4` to slow it down slightly from its default. I should point out the documentation states the allowed rate is between `AVSpeechUtteranceMinimumSpeechRate` and `AVSpeechUtteranceMaximumSpeechRate`. These currently have values of `0.0` and `1.0`, respectively. However, because these are constants, it's possible their values could change in a future iOS release. If you're modifying the `rate` property, it may be safer to calculate the rate as a percentage of the min and max range.
4. Specify the `pitchMultiplier` for the utterance. This changes the pitch of the voice as it speaks this particular utterance. The allowed

values for the `pitchMultiplier` are between 0.5 (low pitch) and 2.0 (high pitch).

5. Specify a `postUtteranceDelay` of 0.1f. This causes the speech synthesizer to pause slightly before speaking the next utterance. You can similarly set a `preUtteranceDelay`.

Run the application and listen to the conversation. It's Hello World done AV Foundation-style!

Experiment with the various `AVSpeechUtterance` settings to get an understanding of how they work. Audition some of the other available voices. Create an instance of `AVSpeechUtterance` with the entire text of *War and Peace* and sit back and relax.

Note

The final versions of iOS 8 and Xcode 6 were released as this book was being finalized. Please see the **Xcode 6 and iOS 8 Notes.pdf** file in the source code repository for additional information on running the sample projects under Xcode 6 and iOS 8.

Summary

This chapter provided you with an introduction to the AV Foundation framework. You should now have a better understanding of where it fits into Apple's media environment and the capabilities it provides. You also now have a better understanding of the digital media domain itself. Although AV Foundation enables you to build some powerful applications without getting too deeply involved in the details of the media, you'll definitely find that the more you understand about the domain, the easier it is to build the applications you desire. AV Foundation is the future of media on Mac OS X and iOS, and this book provides a hands-on guide showing you how to successfully use the framework to build the next generation of media applications.

Challenge

Open AV Foundation's API documentation in either Xcode's documentation browser or on the Apple Developer Connection site. Take some time to browse through the documentation and get a sense for how the classes are

logically related and for the naming conventions used throughout the framework. Doing so will begin to give you a sense of the breadth of capabilities provided by the framework and will better familiarize you with the patterns and conventions used throughout.

2. Playing and Recording Audio

AV Foundation started its life as an audio-only framework. The first incarnation of the framework, introduced in iOS 2.2, contained a single class for handling audio playback; in iOS 3.0, Apple introduced its audio recording counterpart. Although these classes are some of the oldest in the framework, they are still some of the most widely used. In this chapter, we'll look at the `AVAudioPlayer` and `AVAudioRecorder` classes and see how you can leverage these classes to add audio playback and record features to your apps.

Mac and iOS Audio Environments

Before we can begin our discussion of audio playback and recording, it's important that we consider the audio environments of the Mac and iOS platforms. The Mac's audio environment is very free and flexible—and largely under the control of the user. You can simultaneously be playing music from iTunes, watching a QuickTime movie, streaming audio over the Internet, and recording a guitar riff in GarageBand. Although this would almost certainly result in sensory overload, OS X is happy to play along. It actively manages the hardware and software to make this possible but takes a rather passive role in managing the audio user experience.

Let's contrast this with iOS by considering a common scenario:

You begin listening to a song on your iPhone and as the audio plays through its built-in speaker, you receive a phone call. The song quickly fades out, the playback pauses, and you begin to hear the ringtone. You see it's your boss calling to find out if you've been putting the cover sheets on your TPS reports, so you decide to decline the call. The song begins playing again as the audio fades back in. At this point, you realize the iPhone's built-in speaker just isn't doing justice to this classic Parliament-Funkadelic track, so you plug in your headphones. The audio output is routed to the headphones while the track continues to play. When you're done listening, you unplug your headphones, which automatically routes the audio signal back to the built-in speaker and pauses playback.

At a technical level, there is a lot of complexity in the scenario described, but

from the user's perspective it's just magic. iOS provides a *managed audio environment* that adds to the beauty of the overall iOS user experience, but how exactly does this magic happen? This behavior is made possible with the use of an *audio session*.

Understanding Audio Sessions

An audio session acts as an intermediary between your app and the operating system. It provides a simple and elegant way of communicating to the OS how your application should interact with the iOS audio environment. Instead of detailing the specific interactions with the audio hardware, you instead describe the behavior of your app semantically. This enables you to indicate the general audio behavior of your app and delegates the management of this behavior to the audio session so the OS can best manage the user's audio experience.

All iOS applications have an audio session whether they make use of it or not. The default audio session comes preconfigured with the following behavior:

- Audio playback is enabled, but audio recording is disallowed.
- When the user flips the Ring/Silent switch to “silent” mode, any audio being played by your application is muted.
- When the device’s lock screen is presented, the application’s audio is silenced.
- When your application plays audio, any other audio being played in the background is silenced.

The default audio session provides a lot of useful behavior, but it's important to make sure it's the *correct* behavior for your application. The default behavior may work fine for a wide range of applications, but is rarely what's needed when developing media applications. Fortunately, an audio session is easy to tailor to our particular needs with the use of a *category*.

Audio Session Categories

AV Foundation defines seven categories used to describe the audio behavior provided by your application. [Table 2.1](#) lists the available categories and their associated purpose and behaviors.

Category	Purpose	Allows Mixing	Audio Input	Audio Output
Ambient	Games Productivity apps	✓		✓
Solo Ambient (default)	Games Productivity apps			✓
Playback	Audio and video players	Optional		✓
Record	Audio recorder Voice capture		✓	
Play and Record	VoIP, Voice chat	Optional	✓	✓
Audio Processing	Offline conversion and processing			
Multi-Route	Advanced A/V apps making use of external hardware		✓	✓

Table 2.1 **Audio Session Categories**

When determining the appropriate category for your application, you need to ask yourself some questions about its core behavior. Is audio playback an essential or peripheral feature? Should your application's audio be allowed to mix with background audio? Will your application need to capture audio input to record or send across the network? After you've determined your application's essential audio behavior, it becomes easy to choose an appropriate category.

The general behavior provided by a category is sufficient for most applications, but if you need more fine-grained control, some categories can be further customized with the use of *options* and *modes*. Options make it possible for you to enable optional behavior provided by a category, such as whether your application, using the *Playback* category, should allow mixing its audio output with audio playing in the background. Modes further modify a category by introducing behavior tailored to very specific use cases. VoIP and video chat apps frequently make use of modes to get the behavior they require, but there are modes that provide useful behaviors to video playback and recording apps as well.

Configuring an Audio Session

Let's look at an example of how you configure an audio session. An audio

session *can* be modified throughout the life of your application, but you commonly configure it once, upon application start up. A good place to perform this configuration is in the `application:didFinishLaunchingWithOptions:` method inside your application delegate.

[Click here to view code image](#)

```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {  
  
    AVAudioSession *session = [AVAudioSession sharedInstance];  
    NSError *error;  
    if (![session setCategory:AVAudioSessionCategoryPlayback  
error:&error]) {  
        NSLog(@"Category Error: %@", [error  
localizedDescription]);  
    }  
    if (![session setActive:YES error:&error]) {  
        NSLog(@"Activation Error: %@", [error  
localizedDescription]);  
    }  
    return YES;  
}
```

`AVAudioSession` provides the interface to interact with your application's audio session, so you begin by getting a pointer to its singleton instance. You specify that you want an audio session specifically tailored for audio playback by setting the appropriate category. Finally, you tell the audio session to make this configuration active.

We'll look at the details of working with `AVAudioSession` later in this chapter, but let's move on to talk about audio playback.

Audio Playback with `AVAudioPlayer`

Audio playback is a common need for many applications, and AV Foundation makes it easy to implement this functionality thanks to a class called `AVAudioPlayer`. An instance of this class provides you with a simple way of playing audio data either from a file or in memory. Despite its simple interface, it's a highly functional component and is often the best choice for implementing audio playback on Mac and iOS.

`AVAudioPlayer` is built on top of Core Audio's C-based Audio Queue Services. As such, it provides all the core functionality you'll find in Audio

Queue Services, such as playback, looping, and even audio metering, but in a much simpler and friendlier Objective-C interface. Unless you need to play audio from a network stream, need access to the raw audio samples, or require very low latency, AVAudioPlayer is the way to go.

Creating an AVAudioPlayer

An AVAudioPlayer can be constructed in two ways: with an NSData containing an in-memory version of the audio to be played or an NSURL to a local audio file. If you're on iOS, the URL must be a location within the application's sandbox or can additionally be a URL to an item in the user's iPod library. Let's look at an example.

[Click here to view code image](#)

```
NSURL *fileURL =
    [[NSBundle mainBundle] URLForResource:@"rock"
withExtension:@"mp3"];

// Must maintain a strong reference to player
self.player =
    [[AVAudioPlayer alloc] initWithContentsOfURL:fileURL
error:nil];

if (self.player) {
    [self.player prepareToPlay];
}
```

In the example, we're creating an instance of AVAudioPlayer from a URL pointing to "rock.mp3" contained in the application bundle. If a valid player instance is returned, it is recommended you call its `prepareToPlay` method. Doing so will acquire the needed audio hardware and preload the Audio Queue's buffers. Calling `prepareToPlay` is optional and will be implicitly invoked when the `play` method is called, but preparing the player upon creation minimizes the latency between calling the `play` method and hearing audio output.

Controlling Playback

The player instance has the full suite of methods you'd expect to control its playback. Calling its `play` method begins audio playback at the current time, the `pause` method pauses playback, and the `stop` method, you guessed it, stops playback. Interestingly, both the `pause` and `stop` methods have the

same outwardly observable behavior in that they stop the playback at the current time. The next time we invoke the `play` method, after calling either operation, the playback continues at the time stopped or paused. The key difference between these methods is under the hood. Calling the `stop` method will undo the setup performed by the call to `prepareToPlay` whereas calling `pause` will not.

In addition to the standard “transport” methods described previously, you can also take other interesting actions, such as the following:

- **Modify the player’s volume:** The player’s volume is independent of the system volume, which allows us to achieve some interesting effects, such as applying audio fades. The volume or playback gain is defined as a floating-point value ranging from `0.0` (silent) to `1.0` (full volume).
- **Modify the player’s panning, allowing you to position the sound in the stereo field:** The pan value is a `float` ranging from `-1.0` (hard left) to `1.0` (hard right). The default value is `0.0` (center).
- **Adjust the playback rate:** A great addition made in iOS 5 enables you to adjust the playback rate from `0.5` (half speed) to `2.0` (2x speed) without changing the pitch. Slower speeds can be great if you’re transcribing a complex piece of music or spoken word piece, and the fast speeds can be a great way to quickly get through a government regulation seminar.
- **Enable seamless looping of audio by setting the `numberOfLoops` property:** Setting this property to a positive number causes the player to loop n number of times. Alternatively, specifying a value of `-1` causes the player to loop indefinitely.

Audio looping can be performed using both uncompressed, linear PCM audio, as well as compressed audio formats such as AAC. Gapless looping of MP3 clips is possible, but the MP3 format is notoriously challenging as a looping format. MP3s generally need to be prepared specifically for looping purposes using special tools. If you plan to use a compressed format, it is recommended that you use a format such as AAC or Apple Lossless.

- **Perform audio metering:** This enables you to read both the average and peak power levels from a player as playback occurs. You could

feed this data to a VU meter or other visualization component to provide additional visual feedback to the user. We'll dive into this topic when we discuss audio recording later in the chapter.

As you can see, despite its easy-to-use interface, `AVAudioPlayer` is a very capable class and should be the first place you turn for audio playback. Let's dive into the details of `AVAudioPlayer` so we can see (and hear) these features in action.

Building an Audio Looper

In this section you'll create an audio looping application (see [Figure 2.1](#)) that will enable you to synchronize the playback of three player instances, mix those sounds together by controlling each player's volume level and panning within the stereo field, and even control the overall playback rate. You'll find a starter project in the Chapter 2 directory called `AudioLooper_Starter`.

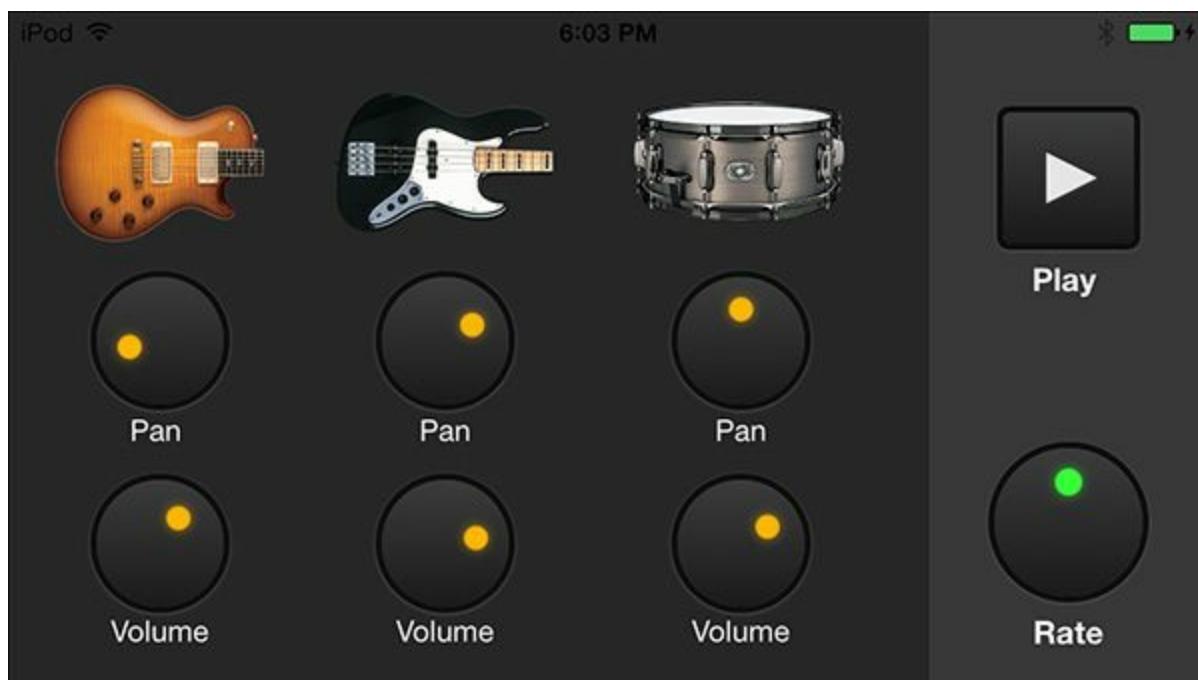


Figure 2.1 Audio Looper app

This sample app, like all others in the book, has a prebuilt user interface with all the actions and outlets wired up and ready to go. The code to manage the audio players and control their playback has been factored into a separate class called `THPlayerController`, enabling you to focus your attention on the AV Foundation implementation. [Listing 2.1](#) shows the interface for this class.

Listing 2.1 THPlayerController Interface

[Click here to view code image](#)

```
@interface THPlayerController : NSObject

@property (nonatomic, readonly, getter = isPlaying) BOOL playing;

// Global methods
- (void)play;
- (void)stop;
- (void)adjustRate:(float)rate;

// Player-specific methods
- (void)adjustPan:(float)pan forPlayerAtIndex:(NSUInteger)index;
- (void)adjustVolume:(float)volume forPlayerAtIndex:
    (NSUInteger)index;

@end
```

The player controller defines `play`, `stop`, and `adjustRate` methods that will aggregate control the playback of the three player instances and additionally defines methods to control the panning and volume levels of the individual players.

If you switch over to the implementation, you'll see you have a stubbed out version of this class. You'll begin implementing the details starting with the initialization section. [Listing 2.2](#) shows the `init` method and a private method used to create the player instances.

Listing 2.2 THPlayerController Initialization

[Click here to view code image](#)

```
#import "THPlayerController.h"
#import <AVFoundation/AVFoundation.h>

@interface THPlayerController ()
@property (nonatomic) BOOL playing;
@property (strong, nonatomic) NSArray *players;
@end

@implementation THPlayerController

- (instancetype)init {
    self = [super init];
    if (self) {
```

```

        AVAudioPlayer *guitarPlayer = [self
playerForFile:@"guitar"];
        AVAudioPlayer *bassPlayer = [self playerForFile:@"bass"];
        AVAudioPlayer *drumsPlayer = [self
playerForFile:@"drums"];
        _players = @[_guitarPlayer, bassPlayer, drumsPlayer];
    }
    return self;
}

- (AVAudioPlayer *)playerForFile:(NSString *)name {
    NSURL *fileURL = [[NSBundle mainBundle] URLForResource:name
withExtension:@"caf"];
    NSError *error;
    AVAudioPlayer *player = [[AVAudioPlayer alloc]
initWithContentsOfURL:fileURL

    if (player) {
        player.numberOfLoops = -1; // loop indefinitely
        player.enableRate = YES;
        [player prepareToPlay];
    } else {
        NSLog(@"Error creating player: %@", [error
localizedDescription]);
    }

    return player;
}

```

Whenever you want to access the classes in the AV Foundation framework, you first need to import `<AVFoundation/AVFoundation.h>`. Thanks to the introduction of *Modules* in Xcode, you no longer have to manually link against the framework; instead, importing the framework automatically does this for you.

Define a class extension and declare a property of type `NSArray` to store the player instances. This will allow you to operate on them as a group, as you'll see shortly.

The application contains three loops: `guitar.caf`, `bass.caf`, and `drums.caf`. These are each four bar loops making up the players in your virtual band. The `init` method creates a player instance for each file by calling the private `playerForFile:` method. This method gets the file's URL and creates a new player instance. Assuming the initialization was successful, it sets its loop count to `-1`, which will cause the player to loop

indefinitely, sets the enableRate property to YES so you can control the playback rate, and finally calls prepareToPlay to prime the player. With the initialization complete, let's move on to the playback methods (See [Listing 2.3](#)).

Listing 2.3 THPlayerController play Method Implementation

[Click here to view code image](#)

```
- (void)play {
    if (!self.playing) {
        NSTimeInterval delayTime = [self.players[0]
deviceCurrentTime] + 0.01;
        for (AVAudioPlayer *player in self.players) {
            [player playAtTime:delayTime];
        }
        self.playing = YES;
    }
}
```

To synchronize the playback of the three player instances, you want to capture the current device time and add a small delay so you have a common reference time from which to begin playback. You loop through your array of players and begin playback by calling playAtTime: on each instance passing in your delayed reference time. This ensures that these players remain tightly in synch as the audio plays. Let's move on and provide an implementation of the stop method as shown in [Listing 2.4](#).

Listing 2.4 THPlayerController stop Method Implementation

[Click here to view code image](#)

```
- (void)stop {
    if (self.playing) {
        for (AVAudioPlayer *player in self.players) {
            [player stop];
            player.currentTime = 0.0f;
        }
        self.playing = NO;
    }
}
```

The stop method is straightforward. If the audio is currently playing, you'll loop through the players array and call stop on each. You'll additionally set

each player's `currentTime` property to `0.0f`, which moves the playhead back to the starting point. Finally, you'll update the playing state to indicate that playback has stopped.

The user interface has a master playback rate knob that enables you to slow down or speed up the playback rate without changing pitch. The rate knob's adjustment has a minimum value of `0.5`, providing half-speed playback, and a maximum value of `1.5`, giving you `1.5x` playback. The `adjustRate:` method takes the incoming value and applies it to each player (see [Listing 2.5](#)).

Listing 2.5 THPlayerController `adjustRate` Method Implementation

[Click here to view code image](#)

```
- (void)adjustRate:(float)rate {
    for (AVAudioPlayer *player in self.players) {
        player.rate = rate;
    }
}
```

Now that you've implemented the global behavior, go ahead and run the app. You can play and stop playback and adjust the rate knob while all three players stay perfectly synchronized.

Note

The knobs on the interface can be reset to their default values by double-tapping on them.

Controlling the Individual Tracks

The app also allows you to adjust the volume and panning of each player. The controls are wired up but currently don't do anything, so let's implement this behavior. You'll implement these methods as shown in [Listing 2.6](#).

Listing 2.6 THPlayerController Volume and Panning Methods

[Click here to view code image](#)

```
- (void)adjustPan:(float)pan forPlayerAtIndex:(NSUInteger)index {
```

```

        if ([self isValidIndex:index]) {
            AVAudioPlayer *player = self.players[index];
            player.pan = pan;
        }
    }

- (void)adjustVolume:(float)volume forPlayerAtIndex:
(NSUInteger)index {
    if ([self isValidIndex:index]) {
        AVAudioPlayer *player = self.players[index];
        player.volume = volume;
    }
}

- (BOOL)isValidIndex:(NSUInteger)index {
    return index == 0 || index < self.players.count;
}

```

The “adjust” methods accept a floating-point adjustment value and an index value identifying the player to be modified. Pan values will be between `-1.0` (hard left) and `1.0` (hard right) and volumes range from `0.0` (silent) and `1.0` (full volume).

Go ahead and run the application again. Now would be a great time to grab your headphones and have some fun twisting the knobs.

The Audio Looper app provides a good example of the utility of `AVAudioPlayer`. You have implemented a complex set of features, but have written a trivial amount of code to do so. If you had written the equivalent code using the underlying Audio Queue Services, both your fingers and brain would be in a lot of pain right now.

Configuring the Audio Session

The Audio Looper app is looking good so far, but let’s run a couple tests to make sure it plays well within the larger iOS audio environment. If you haven’t done so already, build and deploy the app to your device. Begin audio playback and toggle the Ring/Silent switch on the side of your device, and you’ll hear it toggles the audio output. As a second test, while your audio is playing, hit the Lock button on your device. Again, you can hear that the audio is silenced. Because the application’s central purpose is to play audio, neither of these behaviors is what you want or need. This brings us back to the topic of audio sessions.

If you'll recall from our discussion of AVAudioSession in the previous section, every iOS application is automatically provided a default audio session with a category called *Solo Ambient*. This category makes audio playback possible, but is not the appropriate category for applications whose *primary* function is audio playback. Because the default category will not provide the desired behavior, you'll need to explicitly configure the audio session. The audio session can generally be configured once upon app startup, so a good location for this code is in the THAppDelegate's `application:didFinishLaunchingWithOptions:` method.

[Listing 2.7](#) shows how to properly configure Audio Looper's audio session.

Listing 2.7 THAppDelegate Audio Session Setup

[Click here to view code image](#)

```
#import "THAppDelegate.h"
#import <AVFoundation/AVFoundation.h>

@implementation THAppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    AVAudioSession *session = [AVAudioSession sharedInstance];

    NSError *error;
    if (![session setCategory:AVAudioSessionCategoryPlayback
error:&error]) {
        NSLog(@"Category Error: %@", [error
localizedDescription]);
    }
    if (![session setActive:YES error:&error]) {
        NSLog(@"Activation Error: %@", [error
localizedDescription]);
    }

    return YES;
}

@end
```

You begin by getting a handle to the singleton AVAudioSession instance. Because audio playback is the primary functionality, you'll specify a category of AVAudioSessionCategoryPlayback. Finally, you'll

activate the session by calling the `setAction:error:` method passing a value of YES. In this example, we're simply logging any errors that occur, but be sure to handle these appropriately in your production code.

With the audio session properly configured, redeploy the app to your device and run the tests again. You can hear that toggling the Ring/Silent switch no longer mutes the audio. This behavior looks good, so let's verify that your audio continues to play when you lock the device. If you hit the Lock button, you can hear the audio continues to... Houston, we have a problem. You're still seeing the same behavior as before. Shouldn't you have solved this issue by setting the category to `AVAudioSessionCategoryPlayback`? The answer is no. Setting this category makes the app *eligible* to play audio in the background, which is the state we're in when the device locks, but you still need to explicitly opt in to that behavior by making a small change to the application's `Info.plist`.

Find the application's `Info.plist` file (`AudioLooper-Info.plist`) in the Xcode Project Navigator pane and open it in Xcode's property list editor. Add a new key called Required background modes of type Array and add a new entry in this array called App plays audio or streams audio/video using AirPlay (see [Figure 2.2](#)).

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
Localization native development r...	String	en
Bundle display name	String	\${PRODUCT_NAME}
Executable file	String	\${EXECUTABLE_NAME}
Bundle identifier	String	com.tapharmonic.\${PRODUCT_NAME}Identifier
InfoDictionary version	String	6.0
Bundle name	String	\${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1.0
Application requires iPhone envir...	Boolean	YES
▼ Required background modes	Array	(1 item)
Item 0	String	App plays audio or streams audio/video using AirPlay
Main storyboard file base name	String	App plays audio or streams audio/video using AirPlay
► Required device capabilities	Array	App registers for location updates
► Supported interface orientations	Array	App provides Voice over IP services

Figure 2.2 Setting the Required background modes

Alternatively, you can edit the plist in all its XML glory by right-clicking the file in the Project Navigator and selecting Open As > Source Code. Add the

following entry toward the bottom of the file before the closing </dict> tag.

[Click here to view code image](#)

```
<key>UIBackgroundModes</key>
<array>
    <string>audio</string>
</array>
```

Adding this setting specifies that the application is now allowed to play audio in the background. Build and deploy the app again and begin playback. Now if you press the device's Lock button again, you can hear that the groove continues playing in the background. Groovy!

Handling Interruptions

Now that you have the core functionality for the Audio Looper app working, it's time to consider ways to polish the user experience. One of the key ways to polish the behavior is to make sure you're properly handling interruptions. Interruptions are a regular occurrence on an iOS device. In the course of using your device, phone calls come in, alarms go off, and FaceTime requests pop up. Although iOS itself is quite adept at handling these events, we need to make sure *our* handling of these conditions is equally robust.

Let's perform some tests to see how well the Audio Looper app responds to interruptions. To perform the tests, you'll need another phone to call your iPhone or be able to initiate a FaceTime request from another iOS device or from your Mac. Let's perform the following actions:

1. Run the app on your device and begin playback.
2. While the audio is playing, initiate an interruption by either calling your iPhone or starting a FaceTime request from a separate device.
3. Dismiss the call or FaceTime request by tapping the Decline button.

If you were able to run through this test scenario, you observed a couple interesting behaviors. When the interruption began, the audio playback slowly faded out and paused. That happened automatically without writing any code, which is nice. However, when you tapped the Decline button and ended the interruption, you could see and hear a couple problems. The Play/Stop button was left in an invalid state, and the audio playback didn't resume playing as expected. Clearly, this is not the desired behavior, so let's

look at how you can fix these issues.

Audio Session Notifications

Before you can take an action when an interruption occurs, you first need to be notified of its occurrence by registering for a notification posted by the application's AVAudioSession called

AVAudioSessionInterruptionNotification. You'll register for this notification inside the controller's init method, as shown in [Listing 2.8](#).

Listing 2.8 Registering for Interruption Notifications

[Click here to view code image](#)

```
- (instancetype)init {
    self = [super init];
    if (self) {
        AVAudioPlayer *guitarPlayer = [self
playerForFile:@"guitar"];
        AVAudioPlayer *bassPlayer = [self playerForFile:@"bass"];
        AVAudioPlayer *drumsPlayer = [self
playerForFile:@"drums"];

        _players = @[guitarPlayer, bassPlayer, drumsPlayer];

        NSNotificationCenter *nsnc = [NSNotificationCenter
defaultCenter];
        [nsnc addObserver:self
                    selector:@selector(handleInterruption:)
                    name:AVAudioSessionInterruptionNotification
                    object:[AVAudioSession sharedInstance]];

    }
    return self;
}

- (void)dealloc {
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}

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

The posted notification will contain a populated `userInfo` dictionary containing some useful information, enabling you to determine the appropriate action to take.

In the `handleInterruption:` method, you begin by determining the interruption *type* by retrieving the value for the `AVAudioSessionInterruptionTypeKey`. The value returned is an `AVAudioSessionInterruptionType`, which is an enum type indicating whether the interruption began or ended (see [Listing 2.9](#)).

Listing 2.9 Determining the Interruption Type

[Click here to view code image](#)

```
- (void)handleInterruption:(NSNotification *)notification {
    NSDictionary *info = notification.userInfo;
    AVAudioSessionInterruptionType type =
        [info[AVAudioSessionInterruptionTypeKey]
    unsignedIntegerValue];
    if (type == AVAudioSessionInterruptionTypeBegan) {
        // Handle AVAudioSessionInterruptionTypeBegan
    } else {
        // Handle AVAudioSessionInterruptionTypeEnded
    }
}
```

One of the actions to take when interruptions occur is to properly update the state of the Play/Stop button and its associated label.

`THPlayerController` doesn't manage the user interface, so you need a way to communicate those interruption events to the view controller. If you look at the top of the header for `THPlayerController`, you'll see the protocol shown in [Listing 2.10](#).

Listing 2.10 Protocol

[Click here to view code image](#)

```
@protocol THPlayerControllerDelegate <NSObject>
- (void)playbackStopped;
- (void)playbackBegan;
@end
```

The application's view controller already adopts this protocol and is set up as the delegate. This provides a clean and simple way of updating the application's user interface. With that in mind, let's begin by handling the AVAudioSessionInterruptionTypeBegan type in [Listing 2.11](#).

Listing 2.11 Handling Interruption Began

[Click here to view code image](#)

```
- (void)handleInterruption:(NSNotification *)notification {  
  
    NSDictionary *info = notification.userInfo;  
  
    AVAudioSessionInterruptionType type =  
        [info[AVAudioSessionInterruptionTypeKey]  
    unsignedIntegerValue];  
  
    if (type == AVAudioSessionInterruptionTypeBegan) {  
        [self stop];  
        if (self.delegate) {  
            [self.delegate playbackStopped];  
        }  
    } else {  
        // Handle interruption ended  
    }  
}
```

You call the `stop` method and notify the delegate of the interruption state by calling its `playbackStopped` method. It's important to note that by the time this notification is received, the audio session has already been deactivated and the `AVAudioPlayer` instances have been paused. The effect of calling the controller's `stop` is just to update its internal state, not to stop playback.

If the interruption type is `AVAudioSessionInterruptionTypeEnded`, the `userInfo` dictionary will contain an `AVAudioSessionInterruptionOptions` value indicating whether the audio session has been reactivated and playback can resume (see [Listing 2.12](#)).

Listing 2.12 Handling Interruption Ended

[Click here to view code image](#)

```

- (void)handleInterruption:(NSNotification *)notification {
    NSDictionary *info = notification.userInfo;
    AVAudioSessionInterruptionType type =
        [info[AVAudioSessionInterruptionTypeKey]
    unsignedIntegerValue];
    if (type == AVAudioSessionInterruptionTypeBegan) {
        [self stop];
        if (self.delegate) {
            [self.delegate playbackStopped];
        }
    } else {
        AVAudioSessionInterruptionOptions options =
            [info[AVAudioSessionInterruptionOptionKey]
    unsignedIntegerValue];
        if (options ==
    AVAudioSessionInterruptionOptionShouldResume) {
            [self play];
            if (self.delegate) {
                [self.delegate playbackBegan];
            }
        }
    }
}

```

If the options value is equal to

`AVAudioSessionInterruptionOptionShouldResume`, you'll call the controller's `play` method and notify the delegate by calling its `playbackBegan` method.

Build and deploy the application again and rerun the interruption tests. Now when the interruption begins, the player's user interface is properly updated; when control returns to the application, the audio playback begins and everything is in a good state.

Responding to Route Changes

One last item we want to address before we wrap up our coverage of `AVAudioPlayer` is making sure the app properly responds to route changes. A route change happens anytime an audio input or output is added to or removed from an iOS device. This can happen for a variety of reasons, such as a user plugging in a set of headphones or disconnecting a USB

microphone. When these events occur, audio inputs and outputs are rerouted accordingly, and the `AVAudioSession` broadcasts a notification describing the nature of the changes to any interested listeners. To adhere to the behaviors defined in Apple’s Human Interface Guidelines (HIG), the app should become one of those interested listeners.

Let’s perform another test on the Audio Looper app. Begin playback, and while the audio is playing, plug in a set of headphones. The audio output is rerouted to the headphone jack while the audio continues to play, which is precisely the behavior we would expect. With the audio still playing, unplug your headphones. The audio now routes back to the internal speaker and we begin hearing the audio again. Although the routing happened as required, the audio should have been silenced according to Apple’s guidelines. When users plug in a set of headphones, they’re implicitly indicating they want their audio to be private. This means you should continue to respect their privacy when they unplug their headphones by stopping the audio playback. So, how do you do that?

To be notified when routing changes occur, you need to register for a notification posted by the `AVAudioSession` called `AVAudioSessionRouteChangeNotification`. This notification contains a populated `userInfo` dictionary containing a reason indicating why the notification was posted, as well as a previous route description so we can determine the nature of the routing change.

Before you can be notified of these changes, you need to register for the notification. Like you did when registering for interruption notifications, you’ll set up this registration in the `init` method of `THPlayerController` (see [Listing 2.13](#)).

Listing 2.13 Register for Route Change Notifications

[Click here to view code image](#)

```
- (instancetype)init {
    self = [super init];
    if (self) {
        AVAudioPlayer *guitarPlayer = [self
playerForFile:@"guitar"];
        AVAudioPlayer *bassPlayer = [self playerForFile:@"bass"];
        AVAudioPlayer *drumsPlayer = [self
playerForFile:@"drums"];
```

```

    _players = @ [guitarPlayer, bassPlayer, drumsPlayer];

    NSNotificationCenter *nsnc = [NSNotificationCenter
defaultCenter];

    [nsnc addObserver:self
        selector:@selector(handleInterruption:)
        name:AVAudioSessionInterruptionNotification
        object:[AVAudioSession sharedInstance]];

[nsnc addObserver:self
    selector:@selector(handleRouteChange:)
    name:AVAudioSessionRouteChangeNotification
    object:[AVAudioSession sharedInstance]];

}
return self;
}

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

}

```

The first thing to do upon receiving this notification is to determine the reason it occurred. You look up the reason value stored in the `userInfo` dictionary under the `AVAudioSessionRouteChangeReasonKey`. The value returned is an unsigned integer indicating the reason for the change. The reason can indicate a variety of events, such as a new device being attached or a change to the audio session category, but the particular event you're interested in observing is when the headphones have been unplugged. To observe that event you'll watch for a reason called `AVAudioSessionRouteChangeReasonOldDeviceUnavailable` (see [Listing 2.14](#)).

Listing 2.14 Determining the Notification Reason

[Click here to view code image](#)

```

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

    NSDictionary *info = notification.userInfo;

    AVAudioSessionRouteChangeReason reason =
        [info[AVAudioSessionRouteChangeReasonKey]
    unsignedIntValue];

    if (reason ==

```

```
    AVAudioSessionRouteChangeReasonOldDeviceUnavailable) {  
    }  
}
```

After you know a device has been unplugged, you'll ask the `userInfo` dictionary to give you an `AVAudioSessionRouteDescription` describing the previous route. The route description is composed of an `NSArray` of inputs and an `NSArray` of outputs. The elements contained within these arrays are instances of `AVAudioSessionPortDescription`, which describe the attributes of the various I/O ports. In this case you'll look up the first output port from the route description and determine if it's the headphone port. If so, you'll stop playback and call the `playbackStopped` method on the delegate (see [Listing 2.15](#)).

Listing 2.15 Stop Playback on Headphone Unplug

[Click here to view code image](#)

```
- (void)handleRouteChange:(NSNotification *)notification {  
  
    NSDictionary *info = notification.userInfo;  
    AVAudioSessionRouteChangeReason reason =  
        [info[AVAudioSessionRouteChangeReasonKey]  
     unsignedIntValue];  
  
    if (reason ==  
        AVAudioSessionRouteChangeReasonOldDeviceUnavailable) {  
  
        AVAudioSessionRouteDescription *previousRoute =  
            info[AVAudioSessionRouteChangePreviousRouteKey];  
  
        AVAudioSessionPortDescription *previousOutput =  
            previousRoute.outputs[0];  
        NSString *portType = previousOutput.portType;  
  
        if ([portType  
             isEqualToString:AVAudioSessionPortHeadphones]) {  
            [self stop];  
            [self.delegate playbackStopped];  
        }  
    }  
}
```

Build and deploy the latest changes to your device, and test out the new behavior. Now when you unplug your headphones, the audio playback stops just as users will expect.

Handling route changes and properly responding to interruptions are key things we always need to consider when we're building media applications. The little bit of extra effort to correctly handle these scenarios goes a long way in helping you deliver an experience your users will love.

With our Audio Looper now complete, it's time to turn our attention to AVAudioPlayer's counterpart, AVAudioRecorder.

Audio Recording with AVAudioRecorder

You've seen how easy AV Foundation makes it for you to add audio playback features to your apps using the AVAudioPlayer class. AV Foundation makes it equally easy to add audio recording features using the AVAudioRecorder class. AVAudioRecorder, like its audio playback counterpart, is built on top of Audio Queue Services, giving it a great deal of power but in a simple, easy-to-use Objective-C interface. We can use this class to record from the built-in microphones on our Macs or iOS devices as well as record from external audio gear, such as a digital audio interface or USB microphone.

Creating an AVAudioRecorder

You create an instance of AVAudioRecorder by providing it with three pieces of data:

- A local file URL identifying the file into which the audio stream will be written.
- An NSDictionary containing the keys and values used to configure the recording session.
- An NSError pointer used to capture any initialization errors.

Here is how this setup looks in code:

[Click here to view code image](#)

```
NSString *directory = // output directory
NSString *filePath = [directory
    stringByAppendingPathComponent:@"voice.m4a"];
NSURL *url = [NSURL fileURLWithPath:filePath];
```

```

NSDictionary *settings = @{@"AVFormatIDKey" :
@(kAudioFormatMPEG4AAC),
AVSampleRateKey : @22050.0f,
AVNumberOfChannelsKey : @1};

NSError *error;
// Must maintain a strong reference to player
self.recorder =
    [[AVAudioRecorder alloc] initWithURL:url settings:settings
error:&error];
if (self.recorder) {
    [self.recorder prepareToRecord];
} else {
    // Handle error
}

```

Upon successfully creating an instance of `AVRecorder` it is recommended that you call its `prepareToRecord` method. This is similar to `AVPlayer`'s `prepareToPlay` method in that it performs the necessary initialization of the underlying Audio Queue. It additionally creates a file at the location specified by the URL argument to minimize latency when a recording is started.

The keys and values specified in the settings dictionary warrant some discussion. The complete set of valid keys you can define is found in `<AVFoundation/AVAudioSettings.h>`. Many of the keys are specific to a particular format, but the following are common to all audio formats.

Audio Format

The `AVFormatIDKey` key defines the audio format to be written. The following constants are supported as values for the audio format:

[Click here to view code image](#)

```

kAudioFormatLinearPCM
kAudioFormatMPEG4AAC
kAudioFormatAppleLossless
kAudioFormatAppleIMA4
kAudioFormatiLBC
kAudioFormatULaw

```

Specifying `kAudioFormatLinearPCM` results in the uncompressed audio stream being written to the file. This format provides the greatest fidelity but also results in the largest file sizes. Choosing a compressed format such as

AAC (kAudioFormatMPEG4AAC) or Apple IMA4 (kAudioFormatAppleIMA4) results in significantly smaller files and can still provide high quality audio.

Note

The audio format you specify must be compatible with the file type defined in the URL argument. For instance, if you are recording to a file called `test.wav`, you are implicitly indicating that the recorded audio will be in the format required by the *Waveform Audio File Format (WAVE)*, which is little-endian, Linear PCM. Specifying anything other than `kAudioFormatLinearPCM` for the `AVFormatIDKey` value will result in an error. Asking the `NSError` for its `localizedDescription` will return the following error message:

[Click here to view code image](#)

The operation couldn't be completed. (OSStatus error 1718449215.)

The error status of 1718449215 is the integer value for the four-character code, '`fmt?`', which indicates you have defined an incompatible audio format.

Sample Rate

The `AVSampleRateKey` is used to define the recorder's sampling rate. The sampling rate defines the number of samples taken per second of the incoming analog audio signal. The sampling rate plays a significant role in the quality of the audio recording and the size of the resulting file. Low sampling rates, such as 8kHz, will result in a very grainy, AM radio-like recording, but with a very small file size, whereas a 44.1kHz sampling rate (CD-quality sampling) will produce excellent quality, but will result in much larger files. There is no definitive answer on the best sampling rate for your needs, but you should stick with standard sampling rates such as 8,000, 16,000, 22,050, or 44,100. Ultimately, you need to let your ears be the judge.

Number of Channels

The `AVNumberOfChannelsKey` is used to define the number of audio channels recorded. Specifying the default value of 1 will result in a mono

recording, and 2 will result in a stereo recording. Unless you are recording with external hardware, you should generally indicate you will be creating a mono recording.

Format-Specific Keys

You can define additional format-specific keys when working with either Linear PCM or compressed audio formats. See the “AV Foundation Audio Settings Constants” reference in Xcode’s documentation browser for a complete listing.

Controlling Recording

After you’ve created a valid instance of a recorder, you can then begin a new recording. `AVAudioRecorder` has a number of record methods enabling recording for an indefinite period, recording at some point in the future, or recording for a specific duration. You can even pause and then restart a recording, continuing your recording right where you left off. This level of flexibility is quite convenient and makes `AVAudioRecorder` a good choice for many standard recording needs on Mac and iOS.

Now that you have a general understanding of `AVAudioRecorder`, let’s build a simple voice memo application so you can see these features in action.

Building a Voice Memo App

The Voice Memo app (see [Figure 2.3](#)) will enable a user to record a voice memo, pause an in-progress recording, and maintain a list of memos that can be played back. After you get this basic functionality working, we’ll look at a couple ways to further improve the user experience. You’ll find a starter project in the [Chapter 2](#) directory called **VoiceMemo_Starter**.

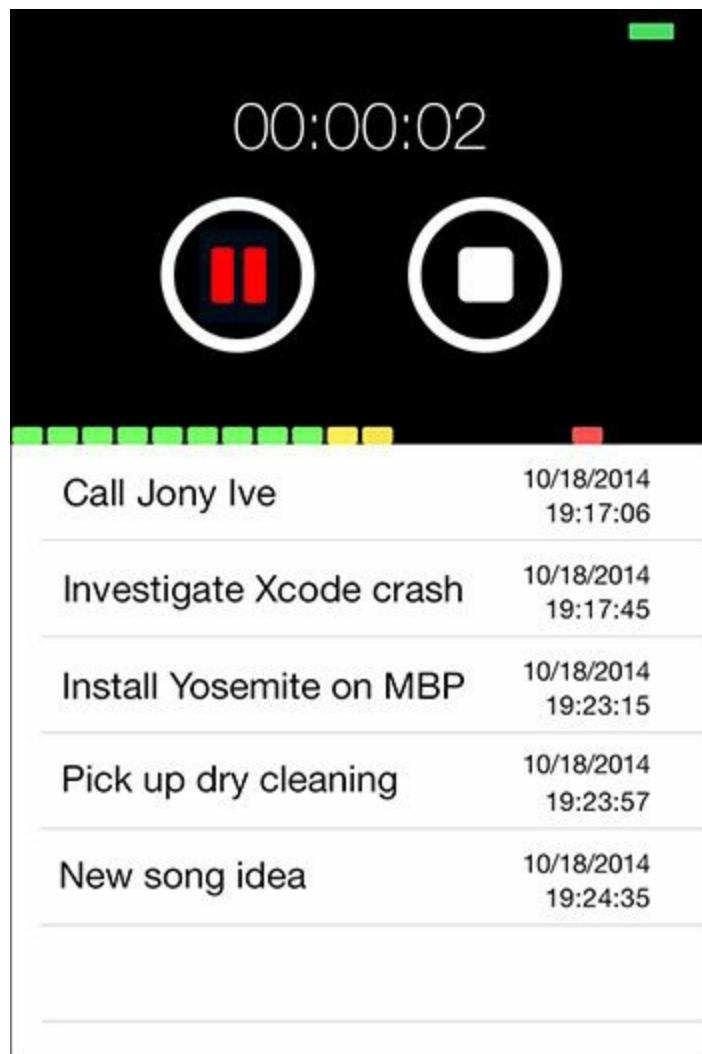


Figure 2.3 Voice Memo app

The app comes with a preconfigured user interface but with no implemented functionality. You'll implement the core recording behavior inside the `THRecorderController` class, but before you do that, let's get the audio session properly configured to enable recording.

Audio Session Configuration

The central purpose of this app is to record and playback voice memos. We know the default category of `Solo Ambient` (`AVAudioSessionCategorySoloAmbient`) won't work, because it doesn't enable audio input. Since you'll need both recording and playback capabilities, the appropriate category for the app will be `AVAudioSessionCategoryPlayAndRecord`. You'll configure the

audio session in the application delegate, THAppDelegate, to use this category as shown in [Listing 2.16](#).

Listing 2.16 THAppDelegate Audio Session Setup

[Click here to view code image](#)

```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {  
  
    AVAudioSession *session = [AVAudioSession sharedInstance];  
  
    NSError *error;  
    if (![session setCategory:AVAudioSessionCategoryPlayAndRecord  
error:&error]) {  
        NSLog(@"Category Error: %@", [error localizedDescription]);  
    }  
  
    if (![session setActive:YES error:&error]) {  
        NSLog(@"Activation Error: %@", [error localizedDescription]);  
    }  
  
    return YES;  
}
```

Starting with iOS 7, the operating system requires a user to explicitly grant permission to an app before it can use the microphone. When an app attempts to access the microphone, the operating system presents the user with a dialog similar to that in [Figure 2.4](#).

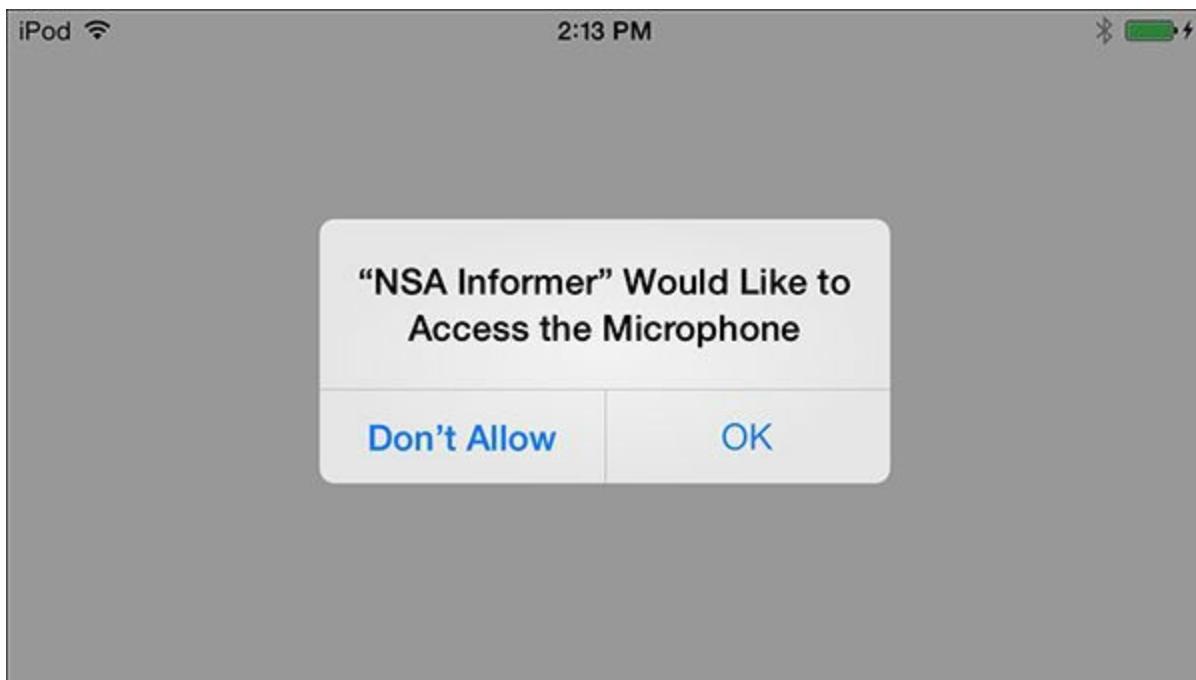


Figure 2.4 iOS requiring user approval before accessing microphone

If the user presses the Don't Allow button, any attempted recording will record only silence. If the user later changes her mind and would like to allow the application to record, she would have to make that change under the Settings app's Privacy > Microphone settings.

Now that you have the audio session configured, let's begin implementing the recorder functionality.

Recorder Implementation

[Listing 2.17](#) shows the interface for the THRecorderController class.

Listing 2.17 **THRecorderController** Interface

[Click here to view code image](#)

```
typedef void(^THRecordingStopCompletionHandler) (BOOL);
typedef void(^THRecordingSaveCompletionHandler) (BOOL, id);

@class THMemo;

@interface THRecorderController : NSObject

@property (nonatomic, readonly) NSString *formattedCurrentTime;

// Recorder methods
```

```

- (BOOL)record;

- (void)pause;

- (void)stopWithCompletionHandler:
(THRecordingStopCompletionHandler)handler;

- (void)saveRecordingWithName:(NSString *)name
completionHandler:
(THRecordingSaveCompletionHandler)handler;

// Player methods
- (BOOL)playbackMemo:(THMemo *)memo;

@end

```

This class defines the core transport operations to control the recording lifecycle as well as methods to manage the playback of previously recorded memos. Let's switch to the class implementation and begin implementing these methods (see [Listing 2.18](#)).

Listing 2.18 THRecorderController Class Extension

[Click here to view code image](#)

```

@interface THRecorderController () <AVAudioRecorderDelegate>

@property (strong, nonatomic) AVAudioPlayer *player;
@property (strong, nonatomic) AVAudioRecorder *recorder;
@property (strong, nonatomic) THRecordingStopCompletionHandler
completionHandler;

@end

```

Inside the `THRecorderController.m` file, you'll define a class extension adopting the `AVAudioRecorderDelegate` protocol. The recorder's delegate protocol defines a method enabling you to be notified when a recording has completed. Knowing that the recording is complete is important so that you can take the opportunity to allow the user to name and save the recording. Let's move on to the controller's `init` method implementation as shown in [Listing 2.19](#).

Listing 2.19 THRecorderController `init` Method

[Click here to view code image](#)

```
- (instancetype)init {
    self = [super init];
    if (self) {
        NSString *tmpDir = NSTemporaryDirectory();
        NSString *filePath = [tmpDir
stringByAppendingPathComponent:@"memo.caf"];
        NSURL *fileURL = [NSURL fileURLWithPath:filePath];

        NSDictionary *settings = @{
            AVFormatIDKey : @(kAudioFormatAppleIMA4),
            AVSampleRateKey : @44100.0f,
            AVNumberOfChannelsKey : @1,
            AVEncoderBitDepthHintKey : @16,
            AVEncoderAudioQualityKey : @(AVAudioQualityMedium)
        };
        NSError *error;
        self.recorder = [[AVAudioRecorder alloc]
initWithURL:fileURL
                                         settings:settings
                                         error:&error];
        if (self.recorder) {
            self.recorder.delegate = self;
            [self.recorder prepareToRecord];
        } else {
            NSLog(@"Error: %@", [error localizedDescription]);
        }
    }
    return self;
}
```

Inside the `init` method you'll perform the recorder configuration. You'll record to the `tmp` directory into a file called `memo.caf`. The Core Audio Format (CAF) is often the best container format to use when recording, because it is content agnostic and may contain any audio format supported by Core Audio. You'll define the recording settings to use Apple IMA4 as the audio format with a sample rate of 44.1kHz, a bit depth of 16-bit, and a single, mono channel. These settings strike a nice balance of quality and file size.

With the audio recorder instance created, let's look at the various transport methods in [Listing 2.20](#).

Listing 2.20 **THRecorderController** Transport Methods

[Click here to view code image](#)

```

- (BOOL)record {
    return [self.recorder record];
}

- (void)pause {
    [self.recorder pause];
}

- (void)stopWithCompletionHandler:
(THRecordingStopCompletionHandler)handler {
    self.completionHandler = handler;
    [self.recorder stop];
}

- (void)audioRecorderDidFinishRecording: (AVAudioRecorder
*)recorder
                                successfully: (BOOL)success {
    if (self.completionHandler) {
        self.completionHandler(success);
    }
}

```

The implementation of these methods is straightforward because you’re just delegating off to the recorder instance to do its work. However, the `stopWithCompletionHandler:` requires a little explanation. When the user taps the Stop button, this method is invoked and passed a completion block. You store a reference to the block and then call the recorder’s `stop` method, which causes the recorder instance to begin finalizing the audio recording. When the audio recording is complete, the recorder invokes its delegate method, at which point you execute the completion block to notify the caller so it can take the appropriate action. In this case, the view controller shows an alert to the user, allowing the user to name and save the recording. Let’s look at the save operation in [Listing 2.21](#).

Listing 2.21 THRecorderController Save Method

[Click here to view code image](#)

```

- (void)saveRecordingWithName: (NSString *)name
                      completionHandler:
(THRecordingSaveCompletionHandler)handler {

    NSTimeInterval timestamp = [NSDate
timeIntervalSinceReferenceDate];
    NSString *filename =

```

```

    [NSString stringWithFormat:@"%@-%f.caf", name, timestamp];

    NSString *docsDir = [self documentsDirectory];
    NSString *destPath = [docsDir
stringByAppendingPathComponent:filename];

    NSURL *srcURL = self.recorder.url;
    NSURL *destURL = [NSURL fileURLWithPath:destPath];

    NSError *error;
    BOOL success = [[NSFileManager defaultManager]
copyItemAtURL:srcURL
toURL:destURL
error:&error];

    if (success) {
        handler(YES, [THMemo memoWithTitle:name url:destURL]);
        [self.recorder prepareToRecord];
    } else {
        handler(NO, error);
    }
}

- (NSString *)documentsDirectory {
    NSArray *paths =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    return [paths objectAtIndex:0];
}

```

When the user has stopped the recording and has been prompted to name the memo, the view controller calls the `saveRecordingWithName:completionHandler:` method. This method copies the recording from the `tmp` directory over to the `Documents` directory with a unique filename. If the copy operation was successful, it calls the completion block, passing back a new instance of `THMemo` containing the name and URL of the audio recording. The view controller code then uses the `THMemo` instance to build its list of recorded memos.

At this point, you can build and run the application and see the core behavior in action. You can record, pause and continue a recording, and stop a recording and save it to your list. Let's look at the last couple of bits of missing functionality.

Although you're able to successfully record and build up a list of memos, you can't currently play them back. You've defined a `playbackMemo:` method

but haven't yet implemented this functionality, so let's do that now. You'll make use of our old friend `AVAudioPlayer` to implement the playback feature (see [Listing 2.22](#)).

Listing 2.22 `THRecorderController` Playback Method

[Click here to view code image](#)

```
- (BOOL)playbackMemo:(THMemo *)memo {
    [self.player stop];
    self.player = [[AVAudioPlayer alloc]
initWithContentsOfURL:memo.url
                           error:nil];
    if (self.player) {
        [self.player play];
        return YES;
    }
    return NO;
}
```

You'll begin by first stopping the existing player (if it exists). You'll then create a new `AVAudioPlayer` instance with the URL stored in the `THMemo` and call the player's `play` method. We're ignoring any initialization errors in this case, but your production code should handle this in a more robust manner.

With the playback behavior working, the last bit of missing functionality you need is related to the time display. The user interface has a label to display the time, but it's permanently fixed at 00:00:00, which is not particularly compelling. `AVAudioRecorder` has a `currentTime` property that makes it easy to build a user interface to provide time feedback to a user. This property returns an `NSTimeInterval`, which indicates the current time in seconds since the beginning of the recording. An `NSTimeInterval` on its own isn't appropriate to display in our user interface, but with a little massaging, you can build something more appropriate. Let's look at the `formattedCurrentTime` method (see [Listing 2.23](#)).

Listing 2.23 `THRecorderController` `formattedCurrentTime`

[Click here to view code image](#)

```
- (NSString *)formattedCurrentTime {
```

```

NSUInteger time = (NSUInteger)self.recorder.currentTime;
NSInteger hours = (time / 3600);
NSInteger minutes = (time / 60) % 60;
NSInteger seconds = time % 60;

NSString *format = @"%02i:%02i:%02i";
return [NSString stringWithFormat:format, hours, minutes,
seconds];
}

```

You start by asking the recorder for its `currentTime`. The returned value is a double type defined as an `NSTimeInterval`. Because you're not interested in the double precision, you'll store the time as an `NSUInteger`. You then calculate the hours, minutes, and seconds available in the current time and create an `NSString` in HH:mm:ss format.

You may be curious about how this is used. This method returns a nicely formatted string suitable for presentation, but it's just a single point in time. How do you keep updating the display showing time ticking by? Your first thought may be to use *Key-Value Observing (KVO)* on the `currentTime` property. Although you will frequently make use of KVO in AV Foundation, you won't when working with `AVAudioRecorder` (and `AVAudioPlayer`), because the `currentTime` property is not observable. Instead, you'll use an `NSTimer` and poll for the value at some periodic interval. [Listing 2.24](#) shows the code from the application's `MainViewController` class that sets up a new `NSTimer` instance that polls every half-second.

Listing 2.24 MainViewController Time Polling

[Click here to view code image](#)

```

- (void)startTimer {
    [self.timer invalidate];
    self.timer = [NSTimer timerWithTimeInterval:0.5
                                             target:self
                                               selector:@selector(updateTi
                                             userInfo:nil
                                             repeats:YES];
    [[NSRunLoop mainRunLoop] addTimer:self.timer
forMode:NSRunLoopCommonModes];
}
- (void)updateTimeDisplay {
    self.timeLabel.text = self.controller.formattedCurrentTime;
}

```

}

Run the application and begin a new recording. The time display now updates, showing you the current time of the recording. If you hit Pause, it will pause at the current time and start right where you left off the next time you tap Record. This provides nice feedback to your users indicating the recording is underway and lets them know how long they've been recording. Although the time display provides some indication that a recording is in progress, it doesn't tell the user if anything is actually being recorded. It would be nice to provide some visual feedback about the audio signal being captured. Fortunately, you can do that by enabling *metering*.

Enabling Audio Metering

A powerful and useful feature available in both `AVAudioRecorder` and `AVAudioPlayer` is the capability to perform audio metering. Audio metering enables you to read both the *average* and *peak* power levels in decibels and use the data to provide additional visual feedback to the user about the audio levels.

The methods available on both classes are called `averagePowerForChannel:` and `peakPowerForChannel:`. Both methods return a floating-point value representing the requested power level in decibels (dB). The values return range from 0dB full scale, which represents maximum power, down to -160dB, which represents the minimum power level or silence.

Before you can read these values, you first need to enable metering by setting the recorder's `meteringEnabled` property to YES. This enables the recorder to perform the power calculations on the captured audio samples. Whenever you want to read the values, you first need to call the `updateMeters` method to get the most recent readings.

In the Voice Memo application, you'd like to enable metering on the `AVAudioRecorder` instance and use this data to display a simple visual meter to the user. The readings provided by the `averagePowerForChannel:` and `peakPowerForChannel:` methods return a floating-point value representing a decibel level, which is a logarithmic unit describing a power ratio. The sample application has a Quartz-based level meter view that displays the average and peak power

levels as the user is recording, but before you can make use of this view the decibel values need to be converted from a *logarithmic* scale of -160 to 0 to a *linear* scale of 0 to 1. You could perform this conversion every time you request the level values; however, a better solution is to calculate these conversions once and look them up as needed. In the sample app, you'll find a class called `THMeterTable` (see [Listing 2.25](#)). This class is a simplified, Objective-C port of Apple's C++-based `MeterTable` class that it uses in a number of its sample projects.

Listing 2.25 THMeterTable Implementation

[Click here to view code image](#)

```
#import "THMeterTable.h"

#define MIN_DB -60.0f
#define TABLE_SIZE 300

@implementation THMeterTable {
    float _scaleFactor;
    NSMutableArray *_meterTable;
}

- (instancetype)init {
    self = [super init];
    if (self) {
        float dbResolution = MIN_DB / (TABLE_SIZE - 1);

        _meterTable = [NSMutableArray
arrayWithCapacity:TABLE_SIZE];
        _scaleFactor = 1.0f / dbResolution;

        float minAmp = dbToAmp(MIN_DB);
        float ampRange = 1.0 - minAmp;
        float invAmpRange = 1.0 / ampRange;

        for (int i = 0; i < TABLE_SIZE; i++) {
            float decibels = i * dbResolution;
            float amp = dbToAmp(decibels);
            float adjAmp = (amp - minAmp) * invAmpRange;
            _meterTable[i] = @(adjAmp);
        }
    }
    return self;
}

float dbToAmp(float dB) {
```

```

        return powf(10.0f, 0.05f * dB);
    }

- (float)valueForPower:(float)power {
    if (power < MIN_DB) {
        return 0.0f;
    } else if (power >= 0.0f) {
        return 1.0f;
    } else {
        int index = (int) (power * _scaleFactor);
        return [_meterTable[index] floatValue];
    }
}

@end

```

This class builds an internal array storing the precomputed decibel to amplitude conversions at some level of decibel resolution, in this case a resolution of -0.2dB. The level of resolution can be adjusted by modifying the `MIN_DB` and `TABLE_SIZE` values.

Each decibel value is converted to its linear scale by calling the `dbToAmp` function, adjusted to fit a range of 0 (-60dB) to 1, and then, to smooth out the curve across the range of values, has its square root calculated and stored in the internal lookup table. These values can then be retrieved by calling the `valueForPower:` method whenever needed.

Let's go back over to the `THRecorderController` class and make the following modifications to make use of this class. You'll add a new property to hold the meter table and create a new instance of it in the `init` method (see [Listing 2.26](#)).

Listing 2.26 **THRecorderController** Meter Table Setup

[Click here to view code image](#)

```

@interface THRecorderController () <AVAudioRecorderDelegate>

@property (strong, nonatomic) AVAudioPlayer *player;
@property (strong, nonatomic) AVAudioRecorder *recorder;
@property (strong, nonatomic) THRecordingStopCompletionHandler
completionHandler;
@property (strong, nonatomic) THMeterTable *meterTable;

@end

```

```

@implementation THRecorderController

- (instancetype)init {
    self = [super init];
    if (self) {
        NSString *tmpDir = NSTemporaryDirectory();
        NSString *filePath = [tmpDir
stringByAppendingPathComponent:@"memo.caf"];
        NSURL *fileURL = [NSURL fileURLWithPath:filePath];

        NSDictionary *settings = @{
            AVFormatIDKey : @(kAudioFormatAppleIMA4),
            AVSampleRateKey : @44100.0f,
            AVNumberOfChannelsKey : @1,
            AVEncoderBitDepthHintKey : @16,
            AVEncoderAudioQualityKey : @(AVAudioQualityMedium)
        };

        NSError *error;
        self.recorder = [[AVAudioRecorder alloc]
initWithURL:fileURL
                                         settings:settings
                                         error:&error];
        if (self.recorder) {
            self.recorder.delegate = self;
            self.recorder.meteringEnabled = YES;
            [self.recorder prepareToRecord];
        } else {
            NSLog(@"Error: %@", [error localizedDescription]);
        }
        _meterTable = [[THMeterTable alloc] init];
    }
    return self;
}

```

Now you'll add a new method to return the levels. [Listing 2.27](#) shows the implementation. Be sure to add the declaration to the class header as well.

Listing 2.27 THRecorderController levels Method

[Click here to view code image](#)

```

- (THLevelPair *)levels {
    [self.recorder updateMeters];
    float avgPower = [self.recorder averagePowerForChannel:0];
    float peakPower = [self.recorder peakPowerForChannel:0];
    float linearLevel = [self.meterTable valueForPower:avgPower];

```

```
    float linearPeak = [self.meterTable valueForPower:peakPower];
    return [THLevelPair levelsWithLevel:linearLevel
peakLevel:linearPeak];
}
```

You begin this method by calling the recorder's `updateMeters` method. This method must be called *immediately prior* to reading the level values to ensure the levels are up to date. You then ask for the average and peak power levels for channel 0. Channels are zero-indexed, and because you're recording in mono, you'll ask for the first channel. You'll then query the meter table for the linear power levels and finally create a new instance of `THLevelPair`. This class already exists in your sample project. It is just a simple value holder to return our average and peak level pair.

Reading the power levels is similar to requesting the current time in that you need to poll the recorder whenever you want to get the latest values. The client code could use an `NSTimer` as it did when requesting the current time. However, because you'll want to update the meter display frequently to keep the animation smooth, an alternative solution would be to use a `CADisplayLink`. A `CADisplayLink` is similar to an `NSTimer`, but is automatically synchronized to the refresh rate of the display. If you open `MainViewController.m`, you'll see the methods shown in [Listing 2.28](#).

Listing 2.28 THMainViewController Metering Methods

[Click here to view code image](#)

```
- (void)startMeterTimer {
    [self.levelTimer invalidate];
    self.levelTimer = [CADisplayLink displayLinkWithTarget:self
                                                    selector:@selector(displayLinkDidFire:)];
    self.levelTimer.frameInterval = 5;
    [self.levelTimer addToRunLoop:[NSRunLoop currentRunLoop]
                           forMode:NSRunLoopCommonModes];
}

- (void)stopMeterTimer {
    [self.levelTimer invalidate];
    self.levelTimer = nil;
    [self.levelMeterView resetLevelMeter];
}

- (void)updateMeter {
    THLevelPair *levels = [self.controller levels];
```

```
    self.levelMeterView.level = levels.level;
    self.levelMeterView.peakLevel = levels.peakLevel;
    [self.levelMeterView setNeedsDisplay];
}
```

When the `startLevelTimer` method is called, it creates a new `CADisplayLink` instance that invokes the `updateLevelMeter` method at some regular interval. By default, that interval is synchronized to the refresh rate, but in this case, I have set the `frameInterval` property to a value of 4, which sets the interval to 1/4 of the refresh rate, which is sufficient for our needs.

The `updateLevelMeter` method queries the `THRecorderController` for its levels, feeds those levels into the `levelMeterView`, and then calls `setNeedsDisplay` causing the view to repaint itself.

Build and run the application again and begin a new recording. The level meter will update its display in response to the volume of your voice. This adds a nice touch to the app because it provides better visual feedback to the user that the memo is actually being recorded.

One thing to note about the meter display, and metering in general, is that it comes at a cost. Enabling metering causes some additional calculations to be performed, which can impact power consumption. Additionally, the level meter in this app is written using Quartz. Quartz is a great framework, but is CPU bound, so it's costly to continuously redraw the meter. Because the application's intent is to record short memos typically no longer than it takes to say "Pick up milk at the grocery store" or "Set up meeting with Charlie for next Friday" this CPU cost probably isn't too great a concern. However, if you intend to record audio for longer periods, you may want to consider disabling metering or switch to a more efficient drawing method, such as using OpenGL ES.

Summary

You saw a great deal of functionality in this chapter provided by AV Foundation's audio-only classes. `AVAudioSession` provides the intermediary between your application and the larger iOS audio environment. It enables you to semantically define the behaviors available to your application using categories and additionally provides the facilities to observe

interruptions and routing changes. AVAudioPlayer and AVAudioRecorder provide simple, yet very powerful interfaces for handling audio playback and recording. Both build on the power of the Core Audio framework but provide a much easier and faster way to add recording and playback capabilities to your app.

3. Working with Assets and Metadata

AV Foundation had fairly humble beginnings. iOS 2.2 and 3.0 brought us the audio-only classes we examined in the previous chapter, and although these can be immensely useful in a variety of applications, they were just a foreshadowing of far greater things to come. It wasn't until iOS 4.0 that AV Foundation exploded into the form it has today. This release brought us a vastly expanded framework with a broad set of capabilities for capturing, composing, playing, and processing media. It also marked a departure from the file-oriented audio classes by centralizing its design around the notion of *assets*. In this chapter you'll begin your journey into the asset-oriented world of AV Foundation by starting with its most important class, `AVAsset`.

Understanding Assets

At the heart of AV Foundation is a class called `AVAsset`. This class is central to AV Foundation's design and plays a key role in virtually all its features and functionality. `AVAsset` is an abstract, immutable class providing a composite representation of a media resource, modeling the static attributes of the media as a whole, such as its title, duration, and metadata. It's important to gain a clear understanding of its purpose and capabilities to successfully leverage AV Foundation.

`AVAsset` abstracts away two important aspects of the media that it models. The first is that it provides a layer of abstraction over the underlying media format. This means whether you are working with a QuickTime movie, an MPEG-4 video, or an MP3 audio track, to you, and to the rest of the framework, it's just an asset. This provides you with a uniform way of working with media without needing to concern yourself with the details of the various codecs and container formats. However, you can always get to those details when you need them. Additionally, `AVAsset` hides the location of the resource. When working with an existing media item you'll create an asset by initializing it with a URL. This could be a local URL pointing to a location in your application bundle or elsewhere on the file system, it could be a URL obtained from the user's iPod Library, or it could even be a URL for an audio or video stream on a remote server. Again, `AVAsset` shields you from these details and leaves the heavy lifting to the framework to ensure

the media is properly retrieved and loaded regardless of its location. By abstracting away the complexities of the media format and location, `AVAsset` provides you with a simple, uniform way of working with timed media.

`AVAsset` is not the media itself, but acts as a container for timed media. It is composed of one or more media tracks along with metadata describing its contents. A class called `AVAssetTrack`, which represents the uniformly typed media contained within the asset, models the individual tracks. An `AVAssetTrack` most commonly models an audio or video stream, but can additionally represent media such as text, subtitles, or closed captions (see [Figure 3.1](#)).



Figure 3.1 `AVAsset` composition

An asset's tracks can be accessed through its `tracks` property. Requesting this property will return an `NSArray` containing all the tracks in its collection. Additionally, `AVAsset` provides the capability to retrieve tracks by identifier, media type, or media characteristic. This enables you to easily retrieve a subset of tracks for further processing.

Creating an Asset

When you create an `AVAsset` for an existing media resource, you do so by initializing it with a URL. This is frequently a local file URL, but could also be a URL for a remote resource.

[Click here to view code image](#)

```
NSURL *assetURL = // url  
AVAsset *asset = [AVAsset assetWithURL:assetURL];
```

`AVAsset` is an abstract class, which means it can't directly be instantiated. When you create an instance using its `assetWithURL:` method, you're actually creating an instance of one of its subclasses called `AVURLAsset`. It can sometimes be useful to use this class directly because it provides you the ability to fine tune how the asset is created by passing it a dictionary of options. For instance, if you are creating an asset that you intend to use in

an audio or video editing scenario, you may want to pass an option to tell it to provide more precise duration and timing, as shown in the following example.

[Click here to view code image](#)

```
NSURL *assetURL = // url
NSDictionary *options =
@{AVURLAssetPreferPreciseDurationAndTimingKey: @YES};
AVURLAsset *asset = [[AVURLAsset alloc] initWithURL:assetURL
options:options];
```

Passing this option provides a hint that you're willing to incur a slightly longer loading time to get more precise duration and timing information.

There are a number of common locations from which you may want to create assets. On an iOS device, you may want to access a video in the user's Photos Library or a song in the iPod Library. On the Mac, you may want to retrieve a media item from the user's iTunes Library. With the help of some ancillary frameworks available on iOS and OS X you can utilize these media resources as well. Let's take a look at a few examples of using these frameworks.

iOS Assets Library

Videos captured by a user using the built-in Camera app or a third-party video capture app are typically written to the user's Photos Library. iOS provides the AssetsLibrary framework that can be used to read and write from this library. The following provides an example of creating an AVAsset from a video in the user's assets library.

[Click here to view code image](#)

```
ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];

[library enumerateGroupsWithTypes:ALAssetsGroupSavedPhotos
                           usingBlock:^(ALAssetsGroup *group, BOOL
*stop) {

    // Filter down to only videos
    [group setAssetsFilter:[ALAssetsFilter allVideos]];

    // Grab the first video returned
    [group enumerateAssetsAtIndexes:[NSIndexSet
indexSetWithIndex:0]
                           options:0
                           usingBlock:^(ALAsset *alAsset,
                           NSUInteger index,
```

```

        BOOL *innerStop) {

            if (alAsset) {
                id representation = [alAsset
defaultRepresentation];
                NSURL *url = [representation url];
                AVAsset *asset = [AVAsset assetWithURL:url];
                // Asset created. Perform some AV Foundation magic.
            }
        ];
    } failureBlock:^(NSError *error) {
        NSLog(@"%@", [error localizedDescription]);
    }];
}

```

This example demonstrates how to retrieve videos stored in the Saved Photos group. It filters that group to just the video content and then retrieves the first video in this collection. Items in this library are modeled as an `ALAsset` object. Asking an `ALAsset` for its default representation will return an `ALAssetRepresentation`, which provides a URL suitable for creating an `AVAsset`.

iOS iPod Library

A common location from which you may want to retrieve media is the user's iPod Library. The MediaPlayer framework provides the API to query for and retrieve items from this library. After the desired item has been found, you can ask for its URL and use it to initialize an asset. Let's look at an example.

[Click here to view code image](#)

```

MPMediaPropertyPredicate *artistPredicate =
[MPMediaPropertyPredicate predicateWithValue:@"Foo Fighters"
forProperty:MPMediaItemPropertyArtist];

MPMediaPropertyPredicate *albumPredicate =
[MPMediaPropertyPredicate predicateWithValue:@"In Your Honor"
forProperty:MPMediaItemPropertyAlbumTitle];

MPMediaPropertyPredicate *songPredicate =
[MPMediaPropertyPredicate predicateWithValue:@"Best of You"
forProperty:MPMediaItemPropertySongTitle];

MPMediaQuery *query = [[MPMediaQuery alloc] init];
[query addFilterPredicate:artistPredicate];
[query addFilterPredicate:albumPredicate];

```

```

[query addFilterPredicate:songPredicate];

NSArray *results = [query items];
if (results.count > 0) {
    MPMediaItem *item = results[0];
    NSURL *assetURL = [item
valueForProperty:MPMediaItemPropertyAssetURL];
    AVAsset *asset = [AVAsset assetWithURL:assetURL];
    // Asset created. Perform some AV Foundation magic.
}

```

The MediaPlayer framework provides a class called `MPMediaPropertyPredicate` that can be used to construct queries enabling you to find any item within the iPod Library. The example code is looking for the song “Best of You,” from the Foo Fighter’s *In Your Honor* album. Upon executing a successful query, it captures the media item’s asset URL property and uses it to create an `AVAsset`.

Mac iTunes Library

On OS X, iTunes is the user’s central repository for media. To identify the resources within this library, developers commonly parse the `iTunes Music Library.xml` file contained within the iTunes music directory and extract the relevant data. Although this is certainly a viable solution, starting with Mac OS X 10.8 and iTunes 11.0, there is now an easier way of working with this library, thanks to the `iTunesLibrary` framework. Let’s see how you can query this library.

[Click here to view code image](#)

```

ITLibrary *library = [ITLibrary libraryWithAPIVersion:@"1.0"
error:nil];

NSArray *items = self.library.allMediaItems;
NSString *query = @"artist.name == 'Robert Johnson' AND "
                  "album.title == 'King of the Delta Blues
Singers' AND "
                  "title == 'Cross Road Blues'";
NSPredicate *predicate =
    [NSPredicate predicateWithFormat:query];

NSArray *songs = [items filteredArrayUsingPredicate:predicate];
if (songs.count > 0) {
    ITLibMediaItem *item = songs[0];
    AVAsset *asset = [AVAsset assetWithURL:item.location];
    // Asset created. Perform some AV Foundation magic.
}

```

}

The iTunesLibrary framework doesn't provide a specific query API like the MediaPlayer framework. However, you can use the standard Cocoa `NSPredicate` class to construct complex queries to find your desired items. After filtering the media items down to your desired set, you can ask an `ITLibMediaItem` for its `location`, which provides a URL suitable for creating an `AVAsset`.

Asynchronous Loading

`AVAsset` has a variety of useful methods and properties to provide information about the asset, such as its duration, creation date, and metadata. It also has methods to retrieve and work with the collection of tracks that it holds. However, it's important to understand that upon creation, an asset is little more than a handle to the underlying media file. `AVAsset` employs a very efficient design that defers loading the properties of an asset until they are requested. This enables you to quickly create assets without immediately paying the penalty of loading its associated media and metadata, but it's important to understand that property access always occurs *synchronously*. If the property you are requesting hasn't previously been loaded, it will block until it's in a state that it can return an appropriate response. Clearly, this can lead to problems. For instance, asking an asset for its `duration` can potentially be an expensive operation. If you're working with an MP3 file that doesn't have a `TLEN` tag set in its header, which defines its duration, the entire audio track will need to be parsed to accurately determine its duration. Assuming this request was issued from the main thread, waiting for this response means you'll be blocking the main thread until the operation completes. In the best-case scenario, you'd be left with a sluggish, unresponsive user interface. In extreme cases on iOS, you may block long enough that the watchdog comes along and terminates your application. This would be an express ticket to a One-Star Review Land. To resolve this issue you should instead be querying an asset's properties *asynchronously*.

`AVAsset` and `AVAssetTrack` both adopt a protocol called `AVAsynchronousKeyValueLoading`. This protocol provides a means of querying properties asynchronously by providing the following methods:

[Click here to view code image](#)

- (AVKeyValueStatus)statusOfValueForKey: (NSString *)key
error: (NSError **)outError
- (void)loadValuesAsynchronouslyForKeys: (NSArray *)keys
completionHandler: (void (^)(void))handler

You can query for the status of a given property using the `statusOfValueForKey:error:` method, which will return an enum value of type `AVKeyValueStatus` indicating the current status of the requested property. If the status is anything other than `AVKeyValueStatusLoaded`, it means it can't be retrieved without potentially blocking. To asynchronously load a given property, you call the `loadValuesAsynchronouslyForKeys:completionHandler:` method, providing it an array of one or more keys (the asset's property names) and a `completionHandler` block that will be invoked when the asset is in a state to answer the request. Let's look at an example.

[Click here to view code image](#)

```
// Create URL for 'sunset.mov' in the application bundle
NSURL *assetURL =
    [[NSBundle mainBundle] URLForResource:@"sunset"
withExtension:@"mov"];

AVAsset *asset = [AVAsset assetWithURL:assetURL];

// Asynchronously load the assets 'tracks' property
NSArray *keys = @[@"tracks"];
[asset loadValuesAsynchronouslyForKeys:keys completionHandler:^{
    // Capture the status of the 'tracks' property
    NSError *error = nil;
    AVKeyValueStatus status =
        [asset statusOfValueForKey:@"tracks" error:&error];

    // Switch over the status to determine its state
    switch (status) {
        case AVKeyValueStatusLoaded:
            // Continue Processing
            break;
        case AVKeyValueStatusFailed:
            // Handle failure with error
            break;
        case AVKeyValueStatusCancelled:
            // Handle explicit cancellation
            break;
    }
}];
```

```
        default:  
            // Handle all other cases  
    }  
};
```

This example creates an `AVAsset` for a QuickTime movie stored in the application bundle and asynchronously loads its `tracks` property. In the completion handler, you want to determine the status of the requested property by calling the asset's `statusOfValueForKey:error:` method. It's important to pass an `NSError` pointer to this method because it will contain error information if the status comes back as `AVKeyValueStatusFailed`. You'll typically switch over the status value and take the appropriate action based on the status returned. Be aware that the completion handler block will be called on an arbitrary queue. Prior to making any updates to your user interface, you must first dispatch back to the main queue.

Note

The example shows loading a single property, `tracks`, but you can request multiple properties in a single call. There are a couple important points to note when you request multiple properties:

1. The `completionHandler` block will be called only once per invocation of `loadValuesAsynchronouslyForKeys:completionHandle`. Its invocation is not tied to the number of keys you pass to this method.
 2. You need to call `statusOfValueForKey:error:` on each property you requested. You can't assume that all properties will return the same status value.
-

Media Metadata

An important concern when building media apps is media organization. Simply presenting a user with a listing of filenames *may* be acceptable for a small handful of files, but that approach clearly won't scale much beyond that. Instead, what you really need is a means of describing the media in a way that enables users to easily find, identify, and organize their media.

Fortunately, all the major media formats you'll work with in AV Foundation provide the capability to embed metadata richly describing their contents. Working with metadata can be a challenging topic because each media type has its own unique format and typically requires a developer to have a low-level understanding of the format to read and write it. Thankfully, AV Foundation makes this much easier by abstracting away most of the format-specific details; it provides you with a relatively uniform way of working with media metadata. In the remainder of this chapter we'll dive into the details of AV Foundation's metadata support to see how you can make use of these facilities in your own apps. Before we get into the specifics, let's take a brief look at how metadata is stored in the various formats you'll encounter.

Metadata Formats

Although a variety of media formats exist, there are really four primary types you encounter in the Apple world: QuickTime (`.mov`), MPEG-4 video (`.mp4` and `.m4v`), MPEG-4 audio (`.m4a`), and MPEG-Layer III audio (`.mp3`).

Although AV Foundation provides a single interface for working with the metadata embedded in these files, there is still value in understanding how and where this data is stored in each type. This will be a fairly high-level discussion, but will provide a basis for further investigation.

QuickTime

QuickTime is a powerful, cross-platform media architecture developed by Apple. Part of that architecture is the specification for the QuickTime File Format that defines the internal structure of a `.mov` file. A QuickTime file is composed of data structures called *atoms*. The general rule is that an atom contains data describing an aspect of the media, or it contains other atoms, but not both. However, there are several cases where this rule is violated by Apple's own implementation. Atoms get composed into a complex tree-like structure that richly describes everything from the layout and format of the audio samples and video frames to the presentable metadata, such as the author and copyright information.

One way to get an understanding of the structure of the QuickTime format is to open a `.mov` file in a hex editor such as Hex Fiend or Synalyze It! Pro. A typical hex tool will provide you with a true representation of a QuickTime file's data, but doesn't make it particularly easy to visualize the structure and

relationships of its atoms. A better solution is to use the *Atom Inspector* tool available from Apple Developer Center. This tool presents the atom structure in an `NSOutlineView` so you can get a better understanding of the hierarchical relationships between atoms, and it also provides a minimal hex viewer so you can see the actual byte layout.

[Figure 3.2](#) provides a simplified illustration of the structure of a QuickTime version of one of my Pixar favorites, *The Incredibles*. A QuickTime file minimally contains three top-level atoms: `ftyp`, describing the file type and compatible brands; `mdat`, containing the actual audio and video media; and the all-important `moov` atom (read moo-vee), which fully describes all the relevant details of the media, including its presentable metadata.

Quicktime Movie (`the_incredibles.mov`)

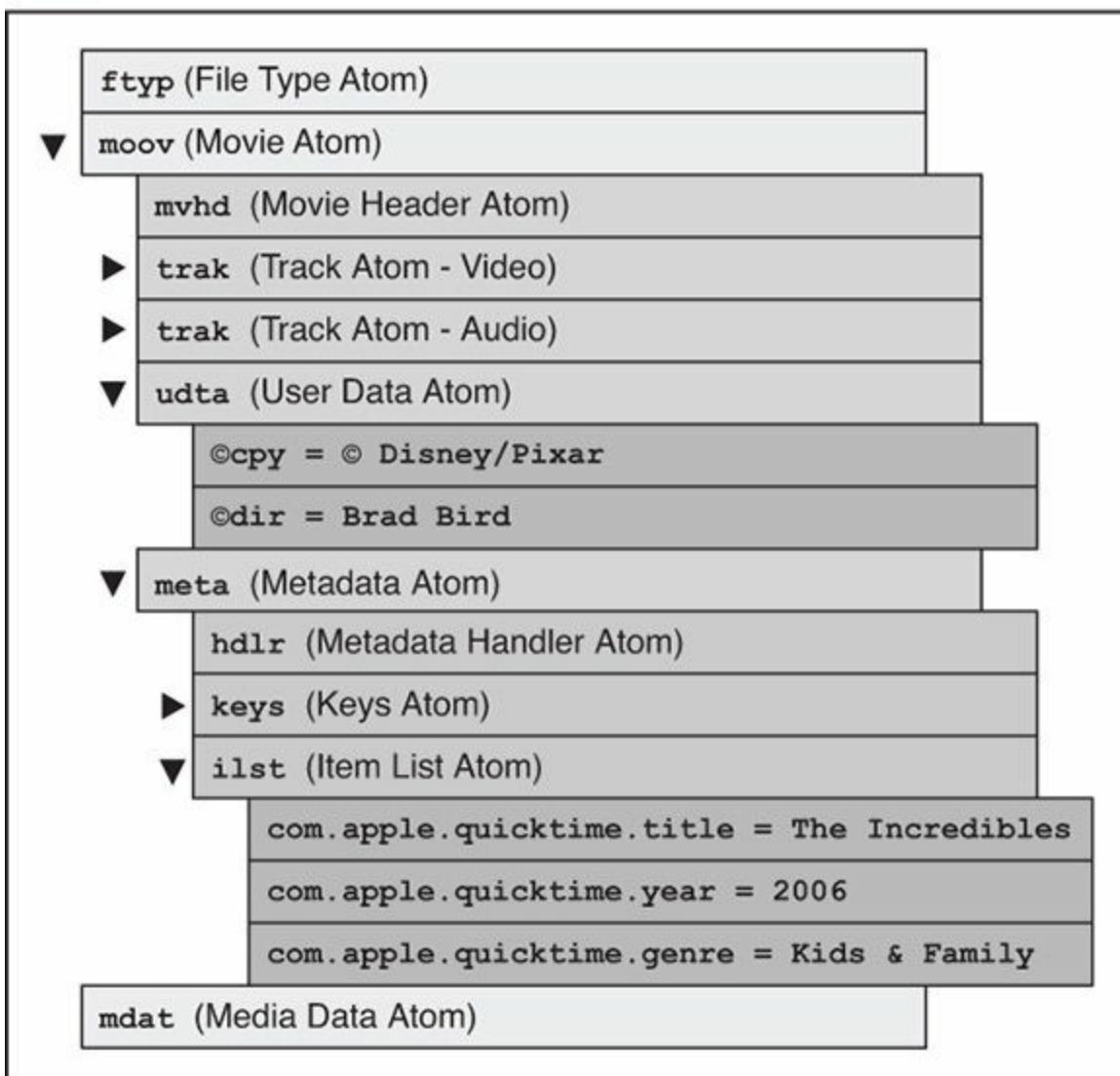


Figure 3.2 QuickTime atom structure

There are two types of metadata you may encounter when working with QuickTime movies. Standard QuickTime metadata, written by a tool such as Final Cut Pro X, will reside under /moov/meta/ilst/ and its keys are almost always prefixed with com.apple.quicktime. The other type is considered QuickTime *user data*, and is stored under /moov/udta/. QuickTime user data can include standard values that players may look for, such as artist or copyright information, but can also include any arbitrary data that may be useful to your application. AV Foundation has the capability to read and write either type of metadata.

Understanding the details and relevance of the various atoms is outside the scope of this discussion. In fact, Apple has a 400+ page document called the QuickTime File Format Specification¹ that fully describes the details of the file format. Although being an expert on the topic is certainly not essential, I would recommend looking at the details of some of the key moov atoms because you can gain some valuable insight into how AV Foundation makes use of this data.

1. QuickTime File Format Specification

—<https://developer.apple.com/library/mac/documentation/quicktime/QTFF/QTFFPreface/qtffPref/>

MPEG–4 Audio and Video

MPEG–4 Part 14 is the specification defining the MP4 file format. MP4 is a direct descendant of the QuickTime file format, which means it has nearly the same structure as a QuickTime file. In fact, you'll often find that tools that know how to parse one file type typically can parse the other (with varying degrees of success). Just like a QuickTime file, an MP4 is composed of data structures called atoms. Technically, the MPEG–4 specification refers to them as *boxes*, but given the format's QuickTime lineage, most developers still refer to them as atoms. [Figure 3.3](#) again shows an illustration of *The Incredibles*, but this time in MP4 format.

MPEG-4 Video (`the_incredibles.m4v`)

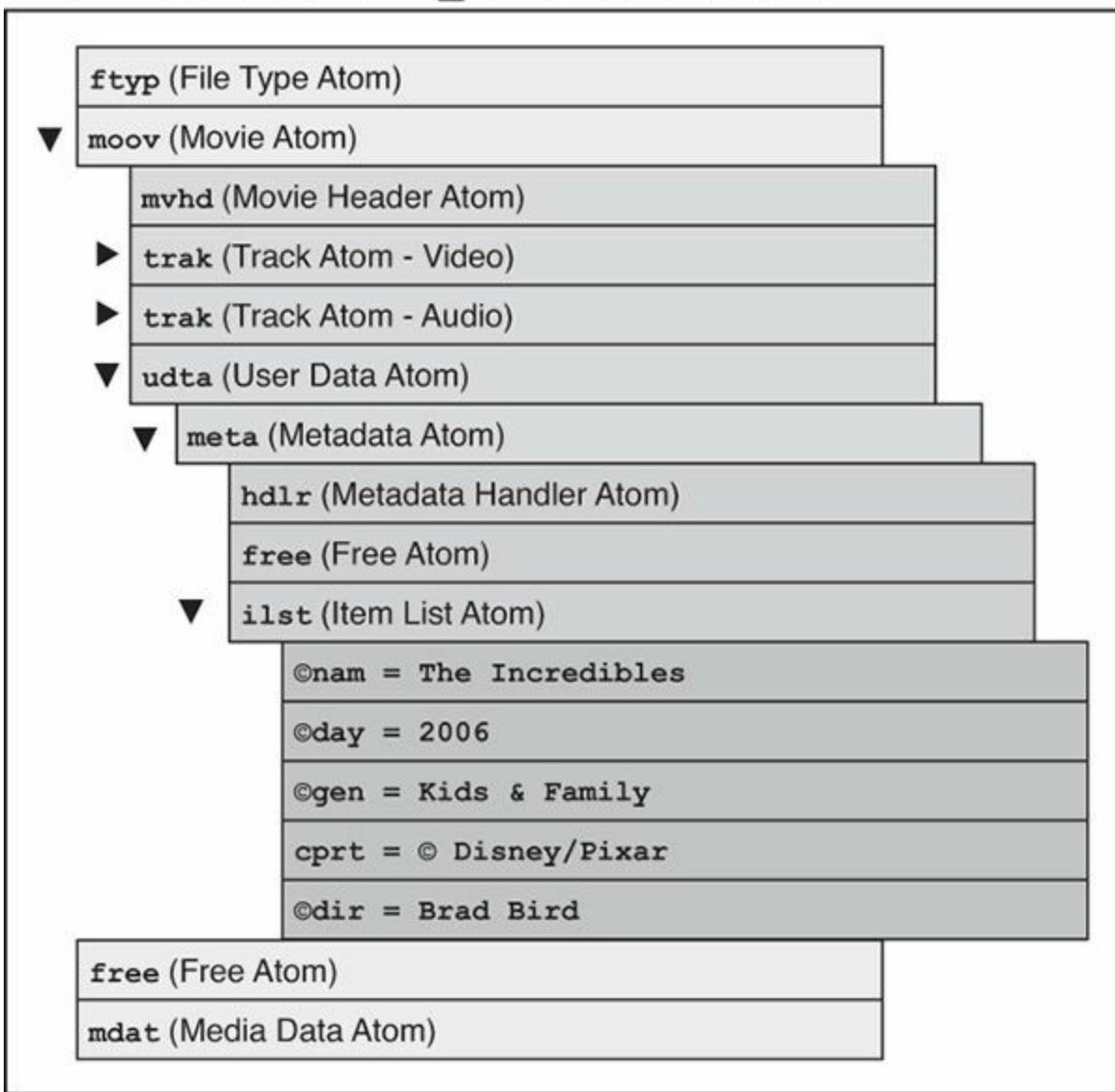


Figure 3.3 MPEG-4 atom structure

The metadata for an MPEG-4 file is found under `/moov/udra/meta/ilst/`. Although there is no standard for the keys you may find under this atom, most tools adopt the keys defined by Apple's unpublished iTunes metadata specification. Despite being unpublished, the iTunes metadata format is widely understood and documented on the Web. One particularly good document on the format can be found from the open-source *mp4v2* library.²

². mp4v2 Library—<https://code.google.com/p/mp4v2/wiki/iTunesMetadata>

Some confusion exists about file extensions. An extension of `.mp4` is the

standard extension defined for MPEG-4 media, but there are variants, such as .m4v, .m4a, .m4p, and .m4b. These variants all use the MPEG-4 container format, but some contain additional extensions. M4V files are MPEG-4 video files with Apple's extensions for FairPlay encryption and AC3-audio. If neither is used, MP4 and M4V vary only in their extensions. M4A is for audio and varies in its file extension as a means of identifying it as an audio-only resource. M4P was Apple's older iTunes audio format using its FairPlay encryption extensions. M4B is used for audio books and typically contains chapter markers and provides the capability to *bookmark* the playback by remembering the user's playback position.

MP3

MP3 files are significantly different from the formats described in the previous sections. An MP3 file does not use a container format; instead, it is the encoded audio data with an optional, structured block of metadata typically prepended to the beginning of the file. MP3 files use a format called ID3v2 to store descriptive information about the audio content, including data such as artist, album, and genre.

Using a hex editor is a good way to investigate the structure of this format. Don't worry, ID3 data is far simpler to read than the atoms described in the previous sections. The first ten bytes of an MP3 file with embedded metadata define the ID3 block's header. The first three bytes of this header will always be '49 44 33' (ID3) indicating this is an ID3v2 tag followed by two bytes that define the major version, either 2, 3, or 4, and a revision number. The remaining bytes define a collection of flags and the ID3 block's size. See [Figure 3.4](#).

ID3 Header				
ID3	Version	Revision	Flags	Size

Figure 3.4 ID3 Header layout

The remaining data in the ID3 block are frames describing the various metadata keys and values. Each frame has a 10-byte header starting with the actual tag name, followed by 4 bytes indicating the size, and then 2 bytes to defining optional flags. The remaining bytes in the frame contain the actual metadata value. If the value is text, which is the usual case, the first byte in the tag defines the type encoding. The type encoding is commonly set to

`0x00`, which indicates ISO-8859-1, but alternate encodings can be supported as well. [Figure 3.5](#) shows an example of the ID3 structure for an MP3 version of John Coltrane’s classic song “Giant Steps.”

MP3 Audio (`giant_steps.mp3`)

ID3 Header				
ID3	Version	Revision	Flags	Size
ID3 Title Frame				
Header		Body		
TIT2	Size	Flags	Enc	Giant Steps
ID3 Artist Frame				
Header		Body		
TPE1	Size	Flags	Enc	John Coltrane
ID3 Genre Frame				
Header		Body		
TCON	Size	Flags	Enc	Jazz
MP3 Data				

Figure 3.5 ID3 structure

AV Foundation supports reading all versions of ID3v2 tags, but does not support writing them. MP3 is a patent-encumbered format, so AV Foundation does not support encoding MP3 or ID3 data.

Note

AV Foundation supports reading all forms of ID3v2 tags, but ID3v2.2 comes with an asterisk. The layout of ID3v2.2 is different from ID3v2.3 and later. Most notably, the individual tags are three characters instead of four. For instance, a song’s *comments*, when

tagged with ID3v2.2, are stored under a COM frame, whereas the same song tagged using ID3v2.3 or greater stores that data under a COMM frame. The string constants defined by the framework are applicable only when working with ID3v2.3 and later. However, you'll see when we get to the sample application how you can still work with ID3v2.2 data.

Working with Metadata

Both AVAsset and AVAssetTrack provide the capability to be queried for their associated metadata. Most commonly you'll utilize the metadata provided by AVAsset, but retrieving track-level metadata can be useful in certain cases. The interface for reading an item's metadata is provided by a class called AVMetadataItem. This class provides an object-oriented interface to access the metadata stored in QuickTime and MPEG-4 atoms and ID3 frames.

AVAsset and AVAssetTrack provide two methods to retrieve the associated metadata. To understand the need for these different methods, you first need to understand the notion of *key spaces*. AV Foundation utilizes key spaces as a way of grouping related keys to enable filtering of collections of AVMetadataItem instances. Every asset will minimally have two key spaces from which metadata is retrieved, as depicted in [Figure 3.6](#).

Common		
iTunes	Quicktime Metadata	ID3
	Quicktime User Data	
 MPEG 4	 MOV	 MP3
iTunes Audio/Video	Quicktime Movie	MPEG Layer III
.mp4, .m4v, .m4a	.mov	.mp3

Figure 3.6 Metadata key spaces

The **common** key space defines keys that are common to all supported media types. This includes common items such as title, artist, and artwork information. This provides some level of metadata normalization across all its supported media formats. You can retrieve metadata from the common key space by asking an asset or track for its `commonMetadata` property, which returns an array of all available common metadata.

Accessing the format-specific metadata is possible by calling the `metadataForFormat:` method on an asset or track. This method takes an `NSString` defining the metadata format and returns an `NSArray` containing all associated metadata. `AVMetadataFormat.h` provides string constants for the various metadata formats it supports. As an alternative to hardcoding a specific metadata format string, an asset or track can also be queried for its `availableMetadataFormats`, which will return an array of strings defining all the metadata formats contained within the resource. You can use this resulting array to iterate through all formats and call `metadataForFormat:` for each. Let's look at an example.

[Click here to view code image](#)

```

NSURL *url = // url
AVAsset *asset = [AVAsset assetWithURL:url];

NSArray *keys = @[@"availableMetadataFormats"];
[asset loadValuesAsynchronouslyForKeys:keys completionHandler:^{
    NSMutableArray *metadata = [NSMutableArray array];
    // Collect all metadata for the available formats
    for (NSString *format in asset.availableMetadataFormats) {
        [metadata addObjectsFromArray:[asset
            metadataForFormat:format]];
    }
    // Process AVMetadataItems
}];

```

Finding Metadata

After you have an array of metadata items, you will typically want to find specific metadata values. One particularly useful way is to use the various convenience methods provided by `AVMetadataItem` to retrieve and filter your result set. For instance, if you're interested in getting the artist and album metadata for a particular M4A audio file, you could retrieve this data as follows:

[Click here to view code image](#)

```

NSArray *metadata = // Collection of AVMetadataItems

NSString *keySpace = AVMetadataKeySpaceiTunes;
NSString *artistKey = AVMetadataiTunesMetadataKeyArtist;
NSString *albumKey = AVMetadataiTunesMetadataKeyAlbum;

NSArray *artistMetadata = [AVMetadataItem
    metadataItemsFromArray:metadata
                           withKey:artistKey
                         keySpace:keySpace];

NSArray *albumMetadata = [AVMetadataItem
    metadataItemsFromArray:metadata
                           withKey:albumKey
                         keySpace:keySpace];

AVMetadataItem *artistItem, *albumItem;
if (artistMetadata.count > 0) {
    artistItem = artistMetadata[0];
}

if (albumMetadata.count > 0) {
    albumItem = albumMetadata[0];
}

```

}

This example makes use of the `metadataItemsFromArray:withKey:keySpace:` method on `AVMetadataItem` to filter the collection down to just those items matching your *key* and *key space* criteria. The return value from this call is an `NSArray`, but will typically contain a single `AVMetadataItem` instance.

Using `AVMetadataItem`

In its most basic form, an `AVMetadataItem` is a wrapper for a key/value pair. You can ask it for its `key` or `commonKey`, if it exists in the common key space, and most importantly its `value`. The `value` property is defined as an `id<NSObject, NSCopying>`, but will be either an `NSString`, `NSNumber`, `NSData`, or in some cases, an `NSDictionary`. If you know the type of the `value` ahead of time, `AVMetadataItem` also provides three type coercion properties—`stringValue`, `numberValue`, and `dataValue`—that simplify typing the return value appropriately.

One common problem most people encounter when first working with `AVMetadataItem` is understanding its `key` property. The `commonKey` is defined as a string and is easy to evaluate against the keys defined in `AVMetadataFormat.h`, but the `key` property is defined as an `id<NSObject, NSCopying>` value. Although this type could, of course, hold an `NSString`, it rarely if ever does. For instance, if I iterate through the embedded metadata in an M4A file in my library with the following code, I'll see some unexpected key values.

[Click here to view code image](#)

```
NSURL *url = // Song URL
AVAsset *asset = // Asset that has its properties loaded

NSArray *metadata =
[asset metadataForFormat:AVMetadataFormatiTunesMetadata];

for (AVMetadataItem *item in metadata) {
    NSLog(@"%@", item.key, item.value);
}
```

Executing this code would produce a listing as follows:

[Click here to view code image](#)

```
-1452383891: Have A Drink On Me
-1455336876: AC/DC
-1451789708: A. Young - M. Young - B. Johnson
-1453233054: Back In Black
-1453039239: 1980
```

Although you can clearly see that the values identify this as an AC/DC song on their *Back in Black* album, what's not clear is the integer values returned for the `key` property. What you're seeing is the integer value of the various key strings. Before you can interpret these values, you need to convert them to their string equivalents. Because this is something you'll do frequently, you can add a category method on `AVMetadataItem` called `keyString` so you can easily retrieve the `NSString` equivalents. Let's look at the implementation of this category method in [Listing 3.1](#).

Listing 3.1 `AVMetadataItem` `keyString` Category Method

[Click here to view code image](#)

```
#import "AVMetadataItem+THAdditions.h"

@implementation AVMetadataItem (THAdditions)

- (NSString *)keyString {
    if ([self.key isKindOfClass:[NSString class]])
    {
        // 1
        return (NSString *)self.key;
    }
    else if ([self.key isKindOfClass:[NSNumber class]]) {
        UInt32 keyValue = [(NSNumber *) self.key
unsignedIntValue];           // 2

        // Most, but not all, keys are 4 characters. ID3v2.2 keys
        // are
        // only be 3 characters long. Adjust the length if
        // necessary.

        size_t length =
        sizeof(UInt32);                                // 3
        if ((keyValue >> 24) == 0) --length;
        if ((keyValue >> 16) == 0) --length;
        if ((keyValue >> 8) == 0) --length;
        if ((keyValue >> 0) == 0) --length;

        long address = (unsigned long)&keyValue;
        address += (sizeof(UInt32) - length);
```

```

    // keys are stored in big-endian format, swap
    keyValue =
CFSwapInt32BigToHost(keyValue);                                // 4

    char
cstring[length];                                              //
5
    strncpy(cstring, (char *) address, length);
cstring[length] = '\0';

    // Replace '@' with '©' to match constants in
AVMetadataFormat.h
    if (cstring[0] == '\xA9')                                     //
6
        cstring[0] = '@';
}

    return [NSString stringWithFormat:(char *)
cstring                                         // 7
encoding:NSUTF8StringEncoding];
}

else {
    return @"<<unknown>>";
}
}

@end

```

- 1.** If the `key` property is already a string, just return it as is. This is an uncommon case.
- 2.** You ask for the `key` as an unsigned integer value. The value returned is a 32-bit, *big endian* number representing a four-character code that you'll extract shortly.
- 3.** In almost all cases the value will be a four-character code such as `©gen` or `TRAK`, but in the case of an MP3 file tagged using ID3v2.2, the key value will be three characters long. The code will shift each byte to determine if the `length` should be shortened.
- 4.** Because the number is in big endian format, you use the `CFSwapInt32BigToHost()` function to flip the byte order to match host CPU, which is *little endian* for both Intel and ARM processors.

5. Create a character array and use the `strncpy` function to populate this array with the character bytes.
6. A large number of QuickTime user data and iTunes keys are prefixed with a © character. However, the keys defined in `AVMetadataFormat.h` are prefixed with an @ symbol. To perform string comparisons against the key constants, you replace the © with a @ character.
7. Finally, you convert the character array into an `NSString` using its `stringWithCString:encoding` initializer.

If you import this category and modify the previous code example to use this new method, you'll see the following, much less cryptic, output:

[Click here to view code image](#)

```
@nam: Have A Drink On Me  
@ART: AC/DC  
@wrt: A. Young - M. Young - B. Johnson  
@alb: Back In Black  
@day: 1980
```

Note

In addition to retrieving an asset's metadata by key and key space, Mac OS X 10.10 and iOS 8 have introduced an additional way to retrieve metadata using an *identifier*. An identifier unifies a key and key space into a single string, providing a slightly simpler model to retrieve an asset's metadata. This chapter uses the key and key space approach because it's compatible across multiple OS versions, but if you're targeting only Mac OS X 10.10 or iOS 8 you may consider using identifiers.

Now that you have a basic understanding of `AVMetadataItem` let's put this knowledge into action by building a Mac metadata editor application called MetaManager.

Building the MetaManager App

The MetaManager app provides a simple interface for viewing and editing metadata (see [Figure 3.7](#)). It enables you to view metadata for all of AV Foundation's supported types and write metadata for all but MP3 files. The

application provides a good example of how to use the metadata classes and also how to deal with the challenges that are often stumbling blocks when you're starting out with the framework. The user interface is ready to go, and you'll instead focus your attention on building out its AV Foundation underpinnings. You'll find a starter copy of this project in the Chapter 3 directory called **MetaManager_starter**.

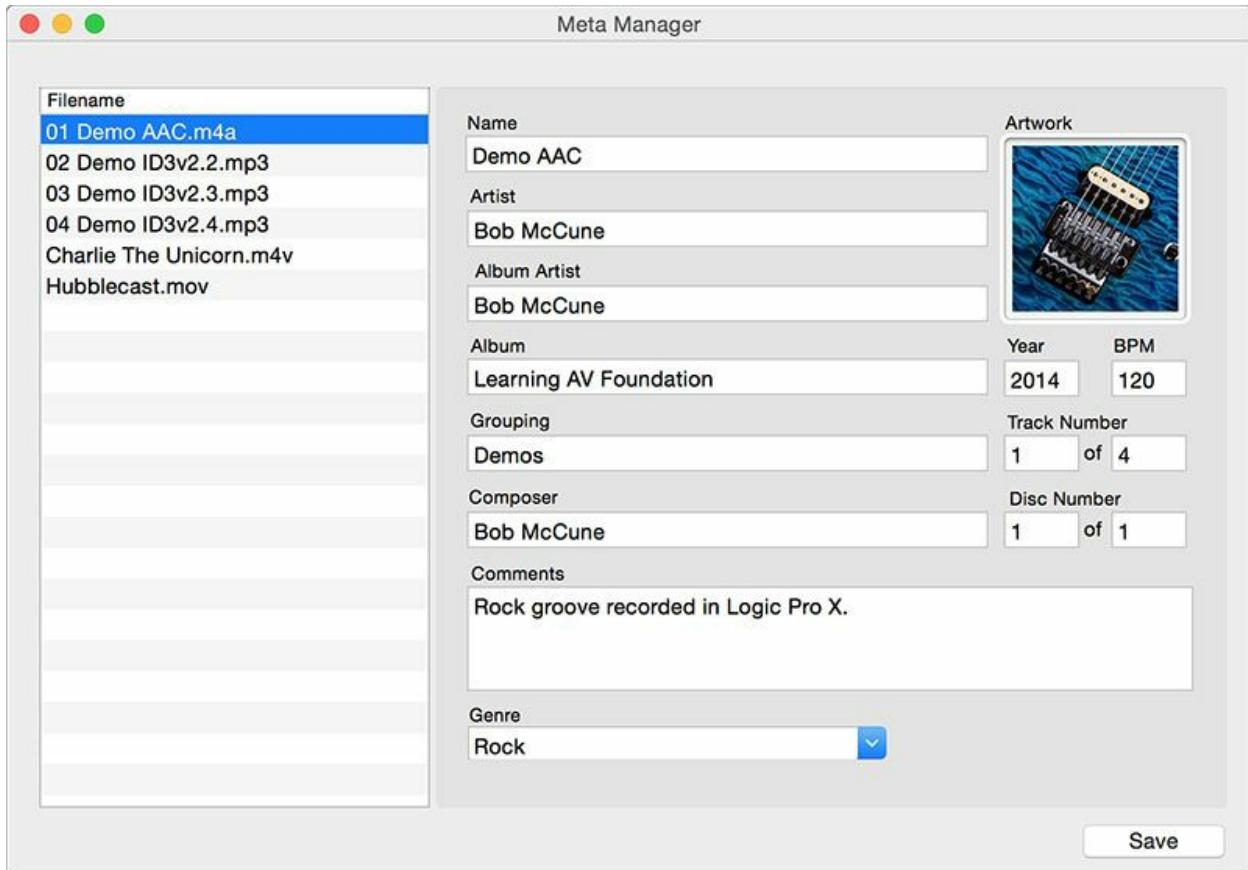


Figure 3.7 MetaManager user interface

Although AV Foundation's metadata classes provide a certain level of abstraction over the underlying metadata formats, it can still be challenging to uniformly manage metadata across a variety of media formats. To provide a consistent interface to the underlying metadata, the strategy you'll employ is to map the format-specific metadata to a normalized set of keys and values. This will provide a simple, consistent client interface to the underlying metadata and will keep the logic to manage the metadata centralized in a single class. Let's dive into the details by looking at the first class you'll be implementing, called `THMediaItem`.

THMediaItem

THMediaItem provides the primary interface for working with the media managed by the application. It provides a wrapper over the underlying AVAsset instance and manages the loading and extraction of the relevant metadata. Let's begin by looking at the interface for this class in [Listing 3.2](#).

Listing 3.2 THMediaItem Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>
#import "THMetadata.h"

typedef void(^THCompletionHandler) (BOOL complete);

@interface THMediaItem : NSObject

@property (copy, readonly) NSString *filename;
@property (copy, readonly) NSString *filetype;
@property (strong, readonly) THMetadata *metadata;
@property (assign, readonly, getter = isEditable) BOOL editable;

- (id)initWithURL:(NSURL *)url;
- (void)prepareWithCompletionHandler:(THCompletionHandler)handler;
- (void)saveWithCompletionHandler:(THCompletionHandler)handler;

@end
```

An instance of THMediaItem is created by calling its `initWithURL:` method, passing it a local file URL for a given media file on disk. The app contains the needed sample files in the application bundle and copies them to `~/Library/Application Support/MetaManager` directory upon application startup. The app will then create an instance of THMediaItem for each file and populate the media list.

Note

If you ever want to reset the media back to its default state, you can go to the Edit menu and select Reset Media Items, which will reset the media to its original state.

Let's switch over to the implementation file and begin implementing its methods, as shown in [Listing 3.3](#).

Listing 3.3 THMediaItem Implementation

[Click here to view code image](#)

```
#import "THMediaItem.h"

#import "AVMetadataItem+THAdditions.h"
#import "NSFileManager+DirectoryLocations.h"

#define COMMON_META_KEY      @"commonMetadata"
#define AVAILABLE_META_KEY   @"availableMetadataFormats"

@interface THMediaItem ()
@property (strong) NSURL *url;
@property (strong) AVAsset *asset;
@property (strong) THMetadata *metadata;
@property (strong) NSArray *acceptedFormats;
@property BOOL prepared;
@end

@implementation THMediaItem

- (id)initWithURL:(NSURL *)url {
    self = [super init];
    if (self) {
        _url =
url;                                              // 1
        _asset = [AVAsset assetWithURL:url];
        _filename = [url lastPathComponent];
        _filetype = [self
fileTypeForURL:url];                           // 2
        _editable = ![_filetype
isEqualToString:AVFileTypeMPEGLayer3];           // 3
        _acceptedFormats =
@[                                                 // 4
            AVMetadataFormatQuickTimeMetadata,
            AVMetadataFormatiTunesMetadata,
            AVMetadataFormatID3Metadata
];
    }
    return self;
}

- (NSString *)fileTypeForURL:(NSURL *)url {
    NSString *ext = [[self.url lastPathComponent] pathExtension];
    NSString *type = nil;
```

```

        if ([ext isEqualToString:@"m4a"]) {
            type = AVFileTypeAppleM4A;
        } else if ([ext isEqualToString:@"m4v"]) {
            type = AVFileTypeAppleM4V;
        } else if ([ext isEqualToString:@"mov"]) {
            type = AVFileTypeQuickTimeMovie;
        } else if ([ext isEqualToString:@"mp4"]) {
            type = AVFileTypeMPEG4;
        } else {
            type = AVFileTypeMPEGLayer3;
        }
        return type;
    }

```

1. In the `initWithURL:` method you set up the internal state of the class and create an instance of `AVAsset` based on the URL passed in. Each item that is displayed in the application's table view is an instance of this class.
2. You determine the file type for the underlying asset. Although AV Foundation provides some more advanced ways to determine the file type, determining the type based on the file extension will suffice. You'll make use of the `filetype` property when you implement the application's save functionality.
3. The `editable` flag will be set based on the media file's type. Although AV Foundation enables you to read ID3 metadata, it does not have the capability to write it, so you'll set the `editable` flag so the application can set the Save button's `enabled` state appropriately.
4. You'll also set up an array of supported metadata formats. This is done to exclude extraneous duplicate values that may show up in the `AVMetadataKeySpaceQuickTimeUserData` and `AVMetadataKeySpaceISOUserData` key spaces.

Let's move on to the implementation of the `prepareWithCompletionHandler:` as shown in [Listing 3.4](#).

Listing 3.4 `prepareWithCompletionHandler:` Implementation

[Click here to view code image](#)

```

- (void)prepareWithCompletionHandler:
(THCompletionHandler)completionHandler {

```

```

        if (self.prepared)
    {
        completionHandler(self.prepared);
        return;
    }

    self.metadata = [[THMetadata alloc]
init]; // 2

    NSArray *keys = @[COMMON_META_KEY, AVAILABLE_META_KEY];

    [self.asset loadValuesAsynchronouslyForKeys:keys
completionHandler:^{

```

AVKeyValueStatus commonStatus =
 [self.asset statusOfValueForKey:COMMON_META_KEY
error:nil];

AVKeyValueStatus formatsStatus =
 [self.asset statusOfValueForKey:AVAILABLE_META_KEY
error:nil];

```

        self.prepared = (commonStatus == AVKeyValueStatusLoaded)
&& // 3
            (formatsStatus == AVKeyValueStatusLoaded);

        if (self.prepared) {
            for (AVMetadataItem *item in
self.asset.commonMetadata) { // 4
                //NSLog(@"%@", item.keyString, item.value);
                [self.metadata addMetadataItem:item
withKey:item.commonKey];
            }

```

for (id format in self.asset.availableMetadataFormats)
{ // 5
 if ([self.acceptedFormats containsObject:format])
{
 NSArray *items = [self.asset
metadataForFormat:format];
 for (AVMetadataItem *item in items) {
 //NSLog(@"%@", item.keyString,
item.value);
 [self.metadata addMetadataItem:item
withKey:item.keyStr:
 }
}
}

}

```
completionHandler(self.prepared);  
6  
}];  
}  
=====
```

1. Each time a user selects a media item in the app's table view, the `prepareWithCompletionHandler:` method gets invoked. You need to prepare this item only once, so if the item is already prepared, invoke the completion block and bail out.
2. You'll create a new instance of `THMetadata`. This class, which you'll be developing shortly, is used to manage the values stored in the individual `AVMetadataItem` instances.
3. Determine the prepared state by evaluating the status of the `commonMetadata` and `availableMetadataFormats` properties. Both properties must be successfully loaded to continue processing.
4. Iterate through all the `AVMetadataItem` instances returned from the common key space and add each to the `THMetadata` instance.
5. Ask the asset for its `availableMetadataFormats`, which will return an array of strings identifying the formats available in the media item. If the particular format is in the collection of `supportedFormats` you'll retrieve the associated `AVMetadataItem` instances and add them to the internal metadata store.
6. Finally, invoke the completion handler block so the user interface can present the retrieved metadata values onscreen.

Go ahead and run the application. It's not yet in a state to present the metadata, but as you select the individual media items, you should see the keys and values being printed to the console. After you've verified this behavior, you can feel free to comment out or completely remove the `NSLog` statements.

Let's move on to the `THMetadata` class and get into the details of working with the values returned from the individual `AVMetadataItem` instances.

THMetadata Implementation

THMetadata is the class performing the bulk of the heavy lifting in this application. It's responsible for extracting the values from the relevant metadata items and storing them for future use. In this class you'll also normalize all the various key values that map to a particular field on the user interface. Let's get started by looking at the interface for THMetadataItem in [Listing 3.5](#).

Listing 3.5 THMetadataItem Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>

@class THGenre;

@interface THMetadata : NSObject

@property (copy) NSString *name;
@property (copy) NSString *artist;
@property (copy) NSString *albumArtist;
@property (copy) NSString *album;
@property (copy) NSString *grouping;
@property (copy) NSString *composer;
@property (copy) NSString *comments;
@property (strong) NSImage *artwork;
@property (strong) THGenre *genre;

@property NSString *year;
@property NSNumber *bpm;
@property NSNumber *trackNumber;
@property NSNumber *trackCount;
@property NSNumber *discNumber;
@property NSNumber *discCount;

- (void)addMetadataItem:(AVMetadataItem *)item forKey:(id)key;
- (NSArray *)metadataItems;

@end
```

The bulk of this interface defines the various data values that will be presented onscreen. Each property maps to a particular field on the user interface. It then defines the `addMetadataItem:` method that adds an `AVMetadataItem` to its internal collection. Finally, it defines a `metadataItems` method that will be used to retrieve the updated metadata so the user can save those changes. Let's switch to the

implementation and get into the details (see [Listing 3.6](#)).

Listing 3.6 THMetadata Implementation

[Click here to view code image](#)

```
#import "THMetadata.h"
#import "THMetadataConverterFactory.h"

@interface THMetadata ()
@property (strong) NSDictionary *keyMapping;
@property (strong) NSMutableDictionary *metadata;
@property (strong) THMetadataConverterFactory *converterFactory;
@end

@implementation THMetadata

- (id)init {
    self = [super init];
    if (self) {
        _keyMapping = [self buildKeyMapping]; // 1
        _metadata = [NSMutableDictionary dictionary]; // 2
        _converterFactory = [[[THMetadataConverterFactory alloc] init]]; // 3
    }
    return self;
}

- (NSDictionary *)buildKeyMapping {

    return @{
        // Name Mapping
        AVMetadataCommonKeyTitle : THMetadataKeyName,
        // Artist Mapping
        AVMetadataCommonKeyArtist : THMetadataKeyArtist,
        AVMetadataQuickTimeMetadataKeyProducer :
        THMetadataKeyArtist,
        // Album Artist Mapping
        AVMetadataID3MetadataKeyBand : THMetadataKeyAlbumArtist,
        AVMetadataiTunesMetadataKeyAlbumArtist :
        THMetadataKeyAlbumArtist,
        @"TP2" : THMetadataKeyAlbumArtist,
        // Continued mappings...
        ...
    };
}
```

}

1. To normalize the keys from the various key spaces and formats, you'll build a mapping from a format-specific key to a normalized key. The normalized key constants are defined in `THMetadataKeys.h`. [Listing 3.6](#) shows an abbreviated version of the `buildKeyMappings` method. Be sure to look at the full version in the source code to get a sense for how all the key values are being mapped.
2. Create an `NSMutableDictionary` to store the display values extracted from the `AVMetadataItem` values. The values stored in this dictionary will be the values presented onscreen.
3. Create an instance of `THMetadataConverterFactory` that will vend instances of `THMetadataConverter`. The converter objects, which you'll be building shortly, are used to convert values between the data stored in an `AVMetadataItem` and the values presented onscreen.

Next, let's take a look at the implementation for the `addMetadataItem:` method (see [Listing 3.7](#)).

Listing 3.7 `addMetadataItem:` Implementation

[Click here to view code image](#)

```
- (void)addMetadataItem:(AVMetadataItem *)item forKey:(id)key {
    NSString *normalizedKey =
    self.keyMapping[key];                                // 1

    if (normalizedKey) {
        id <THMetadataConverter> converter
        =                                         // 2
            [self.converterFactory converterForKey:normalizedKey];

        // Extract the value from the AVMetadataItem instance and
        // convert it into a format suitable for presentation.
        id value = [converter displayValueFromMetadataItem:item];

        // Track and Disc numbers/counts are stored as
        NSDictionary
        if ([value isKindOfClass:[NSDictionary class]])
    }
}
```

```

    {
        // 3
        NSDictionary *data = (NSDictionary *)value;
        for (NSString *currentKey in data) {
            [self setValue:data[currentKey]
forKey:currentKey];
        }
    } else {
        [self setValue:value forKey:normalizedKey];
    }

    // Store the AVMetadataItem away for later use
    self.metadata[normalizedKey] =
item;                                // 4
}

```

1. The first thing you do in this method is retrieve the normalized key value for the key passed into this method. This enables you to map a format-specific key to the normalized key value. If a normalized key was returned, you'd then process the AVMetadataItem.
2. You then ask the converterFactory for a THMetadataConverter object capable of processing the current AVMetadataItem. Ask the converter to provide a display value for the current metadata item, which will extract and convert the value into a format suitable for display.
3. After a value has been returned, you evaluate its type. Disc and track numbers are a special case and require some additional manipulation to extract their values. Ultimately, you use the key-value coding method, setValue:forKey, to set the individual property values.
4. Finally, store the AVMetadataItem in the metadata dictionary so it can be used later.

Don't worry if you're feeling a bit confused by what you've implemented so far. The pieces will begin to fall into place after you start building the converter objects that I've mentioned. Let's move on to building these objects.

Data Converters

One of the most challenging and confusing parts of working with AVMetadataItem is understanding the data stored in its value property.

When the value is a simple string, such as the artist name or album title, or a number, as is the case with the recording year or beats per minute (BPM), the data is easy to understand and requires no real conversion. However, you'll often encounter values that are somewhat obfuscated or just plain cryptic, which means you'll need to go through some extra effort to convert these values into a displayable format. You could put this conversion logic directly into the THMetadata class, but it would quickly become bloated and difficult to maintain. Instead, you'll factor this logic out into a number of converter classes adopting the THMetadataConverter protocol as shown in [Listing 3.8](#).

Listing 3.8 THMetadataConverter Protocol

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>

@protocol THMetadataConverter <NSObject>

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item;

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                         withMetadataItem:(AVMetadataItem *)
                                         item;
@end
```

This protocol defines two methods. The `displayValueFromMetadataItem:` method will be used to extract the data stored in an `AVMetadataItem` instance and convert it into its displayable form. Its counterpart, `metadataItemFromDisplayValue:withMetadataItem:`, will take the display value and convert it back to its `AVMetadataItem` form so it can be persisted back to the underlying media. The class interface for the converter objects will all follow the pattern shown in [Listing 3.9](#), so the subsequent code listings will show only the class implementation.

Listing 3.9 THMetadataConverter Interface Template

[Click here to view code image](#)

```
#import "THMetadataConverter.h"
```

```
@interface ClassName : NSObject <THMetadataConverter>  
@end
```

Simple Conversion

The default implementation of this protocol is simple and will merely provide a pass through to `AVMetadataItem`'s value. This class will be used to convert simple string and number values. [Listing 3.10](#) shows the implementation of this converter.

Listing 3.10 THDefaultMetadataConverter Implementation

[Click here to view code image](#)

```
# import "THDefaultMetadataConverter.h"  
  
@implementation THDefaultMetadataConverter  
  
- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {  
    return item.value;  
}  
  
- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value  
    withMetadataItem:(AVMetadataItem *)item {  
    AVMutableMetadataItem *metadataItem = [item mutableCopy];  
    metadataItem.value = value;  
    return metadataItem;  
}  
  
@end
```

When converting the `AVMetadataItem` value to its display value, you'll return the item's `value` property. This will be sufficient for metadata items whose `value` property is a simple string or number value. When converting from the display value back to an `AVMetadataItem`, you'll begin by creating a mutable copy of the original `AVMetadataItem`, which will return an `AVMutableMetadataItem`. This class is the *mutable* subclass of `AVMetadataItem` you use when creating new or modifying existing metadata. It has the same basic interface as its parent, but it redefines several of its read-only properties as read/write. Creating an `AVMutableMetadataItem` by copying the original `AVMetadataItem`

gives you a new instance with all its important attributes set to the original item's values, leaving you to set its `value` property to the value passed into the method.

This was the simple converter to build, but as you'll see shortly, there are a number of more complex scenarios you'll need to handle. Let's move on to look at how to convert a metadata item's artwork.

Converting Artwork

Media artwork metadata, such as album covers and movie posters, will be returned in a couple different forms. Ultimately, the artwork's bytes are stored in an `NSData` instance, but locating the `NSData` sometimes requires some additional effort. [Listing 3.11](#) provides the implementation for the `THArtworkMetadataConverter`.

Listing 3.11 THArtworkMetadataConverter Implementation

[Click here to view code image](#)

```
#import "THArtworkMetadataConverter.h"

@implementation THArtworkMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {
    NSImage *image = nil;
    if ([item.value isKindOfClass:[NSData class]])
    {
        // 1
        image = [[NSImage alloc] initWithData:item.dataValue];
    }
    else if ([item.value isKindOfClass:[NSDictionary class]])
    {
        // 2
        NSDictionary *dict = (NSDictionary *)item.value;
        image = [[NSImage alloc] initWithData:dict[@"data"]];
    }
    return image;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                         withMetadataItem:(AVMetadataItem *)
*)item {
    AVMutableMetadataItem *metadataItem = [item mutableCopy];

    NSImage *image = (NSImage *)value;
    metadataItem.value =
    image.TIFFRepresentation;                                // 3
}
```

```
    return metadataItem;
}

@end
```

1. If the AVMetadataItem returns a value of type NSData, you create a new NSImage from the bytes stored in this object. Both NSImage and UIImage conveniently provide initializers, allowing you to create an image instance directly from an NSData.
2. If the selected media item is an MP3, you have to dig a little deeper to get the image bytes. The value property in this case will be an NSDictionary, so you cast the value appropriately and then retrieve its "data" key, which will return the NSData containing the image bytes.
3. The implementation to convert the display value into an instance of AVMutableMetadataItem is considerably simpler because AV Foundation can't write ID3 metadata. You can assume the value will be stored as an NSData and ask the NSImage for its TIFFRepresentation. If you would prefer to store the image data as a PNG or JPG, you can use NSBitmapImageRep, the UIImage functions, or the Quartz framework to convert this data as desired.

Converting Comments

Retrieving a media item's comments is fairly straightforward. If you're working with MPEG-4 or QuickTime content, you can ask the AVMetadataItem for its stringValue property. MP3s, again, require a little extra effort to retrieve this data, as shown in [Listing 3.12](#).

Listing 3.12 THCommentMetadataConverter Implementation

[Click here to view code image](#)

```
#import "THCommentMetadataConverter.h"

@implementation THCommentMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {
    NSString *value = nil;
```

```

        if ([item.value isKindOfClass:[NSString class]])
    {
        // 1
        value = item.stringValue;
    }
    else if ([item.value isKindOfClass:[NSDictionary class]])
    {
        // 2
        NSDictionary *dict = (NSDictionary *)item.value;
        if ([dict[@"identifier"] isEqualToString:@""])
        {
            value = dict[@"text"];
        }
    }
    return value;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                         withMetadataItem:(AVMetadataItem
*)item {

    AVMutableMetadataItem *metadataItem = [item
mutableCopy];           // 3
    metadataItem.value = value;
    return metadataItem;
}

@end

```

- 1.** If the item's `value` property is an `NSString`, you can ask for its `stringValue` property. This will be the case for both MPEG-4 and QuickTime media.
- 2.** Comments for an MP3 are stored in an `NSDictionary` defining the ID3 `COMM` frame (`COM` if you're working with ID3v2.2). This frame is a catch-all for a variety of values. For instance, iTunes stores things like its audio normalization and gapless playback settings in this frame, which means you may receive multiple `COMM` frames when requesting ID3 metadata. The particular `COMM` frame containing the actual comments string is contained in the frame with an empty string identifier. After you've found the desired item, you'll retrieve the comments by requesting its `"text"` key.
- 3.** Converting the display value back to an `AVMetadataItem` is easy because you don't need to contend with ID3 data, so you can copy the original `AVMetadataItem` and set its `value` property to the value passed to the method.

Converting Track Data

Audio tracks will commonly contain information indicating a song's ordinal position within a collection of tracks (for example, 4 of 12). MP3 files store this data in an easy-to-understand manner, but M4A is a bit more complex and requires a bit of massaging to produce presentable values. [Listing 3.13](#) shows the implementation for the THTrackMetadataConverter class.

Listing 3.13 THTrackMetadataConverter Implementation

[Click here to view code image](#)

```
#import "THTrackMetadataConverter.h"
#import "THMetadataKeys.h"

@implementation THTrackMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {

    NSNumber *number = nil;
    NSNumber *count = nil;

    if ([item.value isKindOfClass:[NSString class]])
    {
        // 1
        NSArray *components =
            [item.stringValue componentsSeparatedByString:@"/"];
        number = @([components[0] integerValue]);
        count = @([components[1] integerValue]);
    }
    else if ([item.value isKindOfClass:[NSData class]])
    {
        // 2
        NSData *data = item.dataValue;
        if (data.length == 8) {
            uint16_t *values = (uint16_t *) [data bytes];
            if (values[1] > 0) {
                number =
                    @(CFSwapInt16BigToHost(values[1])); // 3
            }
            if (values[2] > 0) {
                count =
                    @(CFSwapInt16BigToHost(values[2])); // 4
            }
        }
    }
}

NSMutableDictionary *dict = [NSMutableDictionary
dictionary]; // 5
[dict setObject:number ?: [NSNull null]
```

```

forKey:THMetadataKeyTrackNumber];
    [dict setObject:count ?: [NSNull null]
forKey:THMetadataKeyTrackCount];

    return dict;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                         withMetadataItem:(AVMetadataItem
*)item {

    AVMutableMetadataItem *metadataItem = [item mutableCopy];

    NSDictionary *trackData = (NSDictionary *)value;
    NSNumber *trackNumber = trackData[THMetadataKeyTrackNumber];
    NSNumber *trackCount = trackData[THMetadataKeyTrackCount];

    uint16_t values[4] =
{0};                                // 6

    if (trackNumber && ![trackNumber isKindOfClass:[NSNull
class]]) {
        values[1] = CFSwapInt16HostToBig([trackNumber
unsignedIntValue]);    // 7
    }

    if (trackCount && ![trackCount isKindOfClass:[NSNull class]])
{
        values[2] = CFSwapInt16HostToBig([trackCount
unsignedIntValue]);    // 8
    }

    size_t length = sizeof(values);
    metadataItem.value = [NSData dataWithBytes:values
length:length];          // 9

    return metadataItem;
}

@end

```

1. MP3 track information will be returned as a string in the form of “xx/xx”. If you’re working with the eighth track of a ten-song album, you’ll receive a string value of “8/10”. In this case you’ll split the value using `NSString’s componentsSeparatedByString:` method to get the individual values and convert them to their integer equivalents and box the values as instances of `NSNumber`.

2. For an M4A file, the value is a bit more cryptic. If you printed the metadata item's value to the console, you'd see it's an `NSData` with a value such as `<00000008 000a0000>`. This is the hex representation for an array of four 16-bit, big endian numbers. The second and third elements in this array contain the track number and track count values, respectively.
3. If the track number is not equal to 0, you'll grab the value and perform an endian swap using the `CFSwapInt16BigToHost()` function to convert it to little endian format and then box the value as an `NSNumber`.
4. Similarly, if the track count value is not equal to 0, you'll grab the value and perform an endian swap on the bytes and wrap the resulting value as an `NSNumber`.
5. Store the captured number and count values in an `NSDictionary` to be returned to the caller. You'll need to check each value to determine if it's `nil`, and if so, replace it with an instance of `NSNull`.
6. Converting the display value back to the form required by `AVMetadataItem` means you'll essentially perform the inverse of the steps you took when extracting those values in the `displayValueFromMetadataItem:` method. You'll create an array of four `uint16_t` values to store the track number and count values.
7. If you have a valid track number, swap the bytes back to big endian format and store in the second position in the array.
8. If you have a valid track count, swap the bytes back to big endian format and store in the third position in the array.
9. Finally, you wrap the `values` array in an instance of `NSData` and set it as the value for the metadata item.

Understanding how to work with track count information certainly isn't straightforward, but now you know the secret!

Converting Disc Data

Disc count information is used to indicate that the CD to which a song belongs is the *nth* disc of *n* discs. Most commonly, the value is *1 of 1*, indicating a single disc, but will have other values if your track's disc is part

of a box set or collection. For instance, if you have Led Zeppelin's *Complete Studio Recordings* (and you really should), the *Led Zeppelin IV* disc would have a value of 4/10. The good news is you already know how to retrieve this data, because it's stored in almost the same way as the track number and track count data from the previous section, so the implementation for this class will look very familiar. [Listing 3.14](#) shows the implementation of the THDiscMetadataConverter class.

Listing 3.14 **THDiscMetadataConverter** Implementation

[Click here to view code image](#)

```
#import "THDiscMetadataConverter.h"
#import "THMetadataKeys.h"

@implementation THDiscMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {

    NSNumber *number = nil;
    NSNumber *count = nil;

    if ([item.value isKindOfClass:[NSString class]])
    {
        // 1
        NSArray *components =
            [item.stringValue componentsSeparatedByString:@"/"];
        number = @([components[0] integerValue]);
        count = @([components[1] integerValue]);
    }
    else if ([item.value isKindOfClass:[NSData class]])
    {
        // 2
        NSData *data = item.dataValue;
        if (data.length == 6) {
            uint16_t *values = (uint16_t *)[data bytes];
            if (values[1] > 0) {
                number =
                    @(CFSwapInt16BigToHost(values[1])); // 3
            }
            if (values[2] > 0) {
                count =
                    @(CFSwapInt16BigToHost(values[2])); // 4
            }
        }
    }

    NSMutableDictionary *dict = [NSMutableDictionary
        dictionaryWithObjectsAndKeys: // 5
            [dict setObject:number forKey:@"number"]
            [NSNull null]
            [dict setObject:count forKey:@"count"]
            [NSNull null]
    ];
}
```

```

forKey:THMetadataKeyDiscNumber];
    [dict setObject:count ?: [NSNull null]
forKey:THMetadataKeyDiscCount];

    return dict;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                         withMetadataItem:(AVMetadataItem
*)item {

    AVMutableMetadataItem *metadataItem = [item mutableCopy];

    NSDictionary *discData = (NSDictionary *)value;
    NSNumber *discNumber = discData[THMetadataKeyDiscNumber];
    NSNumber *discCount = discData[THMetadataKeyDiscCount];

    uint16_t values[3] =
{0};                                // 6

    if (discNumber && (![discNumber isKindOfClass:[NSNull class]]))
{
        values[1] = CFSwapInt16HostToBig([discNumber
unsignedIntValue]);      // 7
    }

    if (discCount && (![discCount isKindOfClass:[NSNull class]])) {
        values[2] = CFSwapInt16HostToBig([discCount
unsignedIntValue]);      // 8
    }

    size_t length = sizeof(values);
    metadataItem.value = [NSData dataWithBytes:values
length:length];           // 9

    return metadataItem;
}

@end

```

1. MP3 disc information will be returned as a string in the form of “xx/xx”. If you’re working with a song from the fourth disc of a ten-disc collection, you’ll receive a string value of “4/10”. In this case you split the value using `NSString’s componentsSeparatedByString:` method to get the individual values and convert them to their integer equivalents and box the values as instances of `NSNumber`.

2. Disc information for an iTunes M4A file is stored as an `NSData`. The `NSData` contains three 16-bit, big endian numbers. The second and third elements in this array contain the disc number and disc count values, respectively.
3. If the disc number is not equal to 0, you'll grab the value and perform an endian swap using the `CFSwapInt16BigToHost()` function to convert it to little endian and then box the value as an `NSNumber`.
4. Similarly, if the disc count value is not equal to 0, you'll grab the value and perform an endian swap on the bytes and wrap the resulting value as an `NSNumber`.
5. Store the captured number and count values in an `NSDictionary` to be returned to the caller. You'll need to check each value to determine if it's `nil`, and if so, replace it with an instance of `NSNull`.
6. Converting the display value back to the form required by `AVMetadataItem` means you'll essentially perform the inverse of the steps you took when extracting those values in the `displayValueFromMetadataItem:` method. You'll create an array of three `uint16_t` values to store the disc number and disc count values.
7. If you have a valid disc number, swap the bytes back to big endian format and store the value in the second position in the array.
8. If you have a valid disc count, swap the bytes back to big endian format and store the value in the third position in the array.
9. Finally, you wrap the values array in an instance of `NSData` and set it as the value for the metadata item.

Converting Genre Data

Working with genre data can be challenging. It's not that the data itself is anything inherently complex, but there are so many forms that genres can take. There are predefined genres, user genres, genre IDs, music genres, movie genres, and so on. Another challenge is that genre information can be stored in more than one way, even for a single file type. Let's talk about some of the basics.

The standard genres used to catalog digital audio originally came from the

MP3 world. The ID3 specification defines 80 default genres and another 46 that are considered WinAmp extensions, for a total of 126 genres. You'll find a number of other commonly used genres that are supported by a variety of tools, but these aren't part of the formal specification.

Given how dominant MP3 was at one point, iTunes, instead of creating its own set of music genres, adopted the ID3 genres, but with a minor twist. iTunes music genres have identifiers that are one greater than their ID3 equivalent. For instance, [Table 3.1](#) shows the first five genres from the ID3 set and the equivalent iTunes values.

ID3 Genre	iTunes Genre
0: Blues	1: Blues
1: Classic Rock	2: Classic Rock
2: Country	3: Country
3: Dance	4: Dance
4: Disco	5: Disco

Table 3.1 Music Genres

Although iTunes adopts the predefined music genres from the ID3 set, it defines its own set of genres for TV shows, movies, and audio books. You can find the complete iTunes genre listing in Apple's Genre IDs Appendix.³

3. Genre IDs Appendix—<http://www.apple.com/itunes/affiliates/resources/documentation/genre-mapping.html>

To simplify working with genre data (and to save you some typing), the sample app includes a class called THGenre that defines the standard music genres as well as the genres used for TV shows. You'll use these genres for the application's audio and video content.

Let's get into the details of working with genres. [Listing 3.15](#) shows the implementation of the THGenreMetadataConverter class.

Listing 3.15 THGenreMetadataConverter Implementation

[Click here to view code image](#)

```
#import "THGenreMetadataConverter.h"
#import "THGenre.h"

@implementation THGenreMetadataConverter
```

```

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {
    THGenre *genre = nil;

    if ([item.value isKindOfClass:[NSString class]])           // 1
    {
        if ([item.keySpace isEqualToString:AVMetadataKeySpaceID3])
        {
            // ID3v2.4 stores the genre as an index value
            if (item.numberValue)
            {
                NSUInteger genreIndex = [item.numberValue
unsignedIntValue];
                genre = [THGenre id3GenreWithIndex:genreIndex];
            } else {
                genre = [THGenre
id3GenreWithName:item.stringValue];                      // 3
            }
        } else {
            genre = [THGenre
videoGenreWithName:item.stringValue];                  // 4
        }
    }
    else if ([item.value isKindOfClass:[NSData class]])      // 5
    {
        NSData *data = item.dataValue;
        if (data.length == 2) {
            uint16_t *values = (uint16_t *)[data bytes];
            uint16_t genreIndex = CFSwapInt16BigToHost(values[0]);
            genre = [THGenre itunesGenreWithIndex:genreIndex];
        }
    }
    return genre;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
    withMetadataItem:(AVMetadataItem *)
    *item {
    AVMutableMetadataItem *metadataItem = [item mutableCopy];

    THGenre *genre = (THGenre *)value;

    if ([item.value isKindOfClass:[NSString class]])           // 6
    {
        metadataItem.value = genre.name;
    }
    else if ([item.value isKindOfClass:[NSData class]])       // 7
    {
        NSData *data = item.dataValue;

```

```

        if (data.length == 2) {
            uint16_t value = CFSwapInt16HostToBig(genre.index +
1);
            size_t length = sizeof(value);
            metadataItem.value = [NSData dataWithBytes:&value
length:length];
        }
    }

    return metadataItem;
}

@end

```

1. Several formats will store the genre data as an `NSString`. Sometimes this string is the actual genre name and sometimes it's an index into a predefined genre list.
2. If the metadata item was pulled from the ID3 key space and its value can be coerced into an `NSNumber`, which is the case when working with ID3v2.4, you'll grab its value as an unsigned integer value and use that index to look up the appropriate `THGenre` instance.
3. The other variants of ID3 will store the genre as the actual name, such as Jazz, Rock, Country, and the like. If the value is the genre name, you'll use that name to find the corresponding `THGenre` instance.
4. For all other genres stored as an `NSString` you'll make the assumption you are working with either a QuickTime movie or an MPEG-4 video file, both of which store the genre as the actual genre name. You'll look up the appropriate video genre from the `THGenre` class.
5. When using one of the predefined genres, an iTunes M4A audio will return its genre as a 16-bit, big endian number stored in an `NSData`. You'll grab its value and swap the bytes to little endian format and then call the `iTunesGenreWithIndex:` method to return the appropriate genre instance.
6. When converting back to `AVMetadataItem` format, if the original `AVMetadataItem` stored its value as a string, simply set the incoming value as the item's `value` property.
7. If the original `AVMetadataItem` stored its value as an `NSData`,

convert the display value back to the appropriate format by swapping the bytes back to big endian format and wrapping the resulting value as an `NSData`. Also note that `THGenre`'s `index` property is zero-based, so to adjust it back to the iTunes equivalent you'll add 1 to its value.

Congratulations, your converter creation is complete! You can now run the application and select items from the list and see their values displayed in the user interface. The one remaining feature you have to implement is adding the capability for users to save their changes. Let's switch back to the `THMetadata` class and finish its implementation.

Finalizing `THMetadata`

The one remaining method to be implemented on `THMetadata` is its `metadataItems` method. This method takes all the display values that are currently stored and converts them back into `AVMetadataItem` format using the converter classes you just finished creating. [Listing 3.16](#) provides the implementation for this method.

Listing 3.16 `metadataItems` Method Implementation

[Click here to view code image](#)

```
- (NSArray *)metadataItems {
    NSMutableArray *items = [NSMutableArray
array]; // 1

    // Add track number/count if applicable
    [self
addMetadataItemForNumber:self.trackNumber
2
        count:self.trackCount
        numberKey:THMetadataKeyTrackNumber
        countKey:THMetadataKeyTrackCount
        toArray:items];

    // Add disc number/count if applicable
    [self addMetadataItemForNumber:self.discNumber
        count:self.discCount
        numberKey:THMetadataKeyDiscNumber
        countKey:THMetadataKeyDiscCount
        toArray:items];

    NSMutableDictionary *metaDict = [self.metadata
```

```

mutableCopy]; // 5
[metaDict removeObjectForKey:THMetadataKeyTrackNumber];
[metaDict removeObjectForKey:THMetadataKeyDiscNumber];

for (NSString *key in metaDict) {

    id <THMetadataConverter> converter =
        [self.converterFactory converterForKey:key];

    id value = [self
valueForKey:key]; // 6

    AVMetadataItem *item
    = // 7
        [converter metadataItemFromDisplayValue:value
withMetadataItem:metaDict[key]].

    if (item) {
        [items addObject:item];
    }
}

return items;
}

- (void)addMetadataItemForNumber: (NSNumber *) number
                           count: (NSNumber *) count
                         numberKey: (NSString *) numberKey
                        countKey: (NSString *) countKey
                          toArray: (NSMutableArray *) items {

    id <THMetadataConverter> converter =
        [self.converterFactory converterForKey:numberKey];

    NSDictionary *data = @{@"numberKey" : number ?: [NSNull
null], // 3
                           "countKey" : count ?: [NSNull null]};

    AVMetadataItem *sourceItem = self.metadata[numberKey];

    AVMetadataItem *item
    = // 4
        [converter metadataItemFromDisplayValue:data
withMetadataItem:sourceItem];
    if (item) {
        [items addObject:item];
    }
}

```

1. You create an instance of `NSMutableArray` to store the collection of metadata items this method will build.
2. Both the track number/count and disc number/count require some additional processing to convert their values back into an `AVMetadataItem` format so that logic will be factored out into a separate method, as shown in #3.
3. You wrap the number and count pair in an `NSDictionary` and grab the original `AVMetadataItem` to be passed to the converter.
4. You retrieve the metadata item from the converter, and if it returned a valid metadata instance, you'll add it to the `items` array.
5. Create a copy of the internal `metadata` dictionary containing the various display values and remove the track and disc number keys from the array, because these have already been processed. You'll iterate over the remaining keys and values to create the appropriate metadata items.
6. Use the key/value coding method `valueForKey:` to look up the value associated with the current key.
7. Finally, retrieve the `AVMetadataItem` instance from the current converter, and if a valid instance was returned, add it to the `items` collection.

`THMetadata` is now complete, and you've almost completed the `MetaManager` application. The only thing remaining is to provide an implementation for `saveWithCompletionHandler:` method on `THMediaItem`. Let's do that now.

Saving Metadata

You've built out the `THMetadata` class and its associated converter objects, so you can now read metadata and convert the user's input back into `AVMetadataItem` instances. However, one important question remains. Because `AVAsset` is an immutable class, how do you apply those metadata changes? The answer is that you *don't* directly modify an `AVAsset`, but instead use a class called `AVAssetExportSession` to export a new copy of the asset along with the metadata changes. Before we get into the details of implementing this behavior, let's take a brief look at configuring and using an

`AVAssetExportSession`.

Using `AVAssetExportSession`

An `AVAssetExportSession` is used to transcode the contents of an `AVAsset` according to an export preset and then write the exported asset to disk. It provides a number of features that enable you to convert from one format to another, trim the contents of an asset, modify an asset's audio and video behavior, and, as you're most interested in at the moment, write new metadata.

You create an instance of `AVAssetExportSession` by providing it a source asset and an export preset. The export preset is used to determine features such as the quality and scaling of the exported content. After you've created an export session, you also need to specify an `outputURL` declaring where the exported content will be written to and an `outputFileType` indicating the output format that will be written. Finally, you call its `exportAsynchronouslyWithCompletionHandler:` method to begin the export process.

Instead of talking about `AVAssetExportSession` in abstract terms, let's put it into action in the `saveWithCompletionHandler:` method. [Listing 3.17](#) provides the implementation for this method.

Listing 3.17 `THMediaItem` `saveWithCompletionHandler:` Implementation

[Click here to view code image](#)

```
- (void)saveWithCompletionHandler:(THCompletionHandler)handler {
    NSString *presetName =
    AVAssetExportPresetPassthrough; // 1
    AVAssetExportSession *session =
        [[AVAssetExportSession alloc] initWithAsset:self.asset
                                         presetName:presetName];

    NSURL *outputURL = [self
tempURL]; // 2
    session.outputURL = outputURL;
    session.outputFileType = self.filetype;
    session.metadata = [self.metadata
metadataItems];
    // 3
```

```

[session exportAsynchronouslyWithCompletionHandler:^{
    AVAssetExportSessionStatus status = session.status;
    BOOL success = (status ==
AVAssetExportSessionStatusCompleted);
    if (success)
    {
        // 4
        NSURL *sourceURL = self.url;
        NSFileManager *manager = [NSFileManager
defaultManager];
        [manager removeItemAtURL:sourceURL error:nil];
        [manager moveItemAtURL:outputURL toURL:sourceURL
error:nil];
        [self
reset];
    }
    if (handler) {
        handler(success);
    }
}];
}

- (NSURL *)tempURL {
    NSString *tempDir = NSTemporaryDirectory();
    NSString *ext = [[self.url lastPathComponent] pathExtension];
    NSString *tempName = [NSString stringWithFormat:@"temp.%@", ext];
    NSString *tempPath = [tempDir
stringByAppendingPathComponent:tempName];
    return [NSURL fileURLWithPath:tempPath];
}

- (void)reset {
    _prepared = NO;
    _asset = [AVAsset assetWithURL:self.url];
}

```

- 1.** Begin by creating an `AVAssetExportSession` using the `AVAssetExportPresetPassthrough` preset.
`AVAssetExportSession` has a number of presets available, but this preset enables you to write the new metadata without incurring the overhead of re-encoding the media. A “passthrough” export happens in a fraction of the time it would take if you were to transcode the actual media content.
- 2.** Create a URL for the temporary file that will be written to disk. This URL is based on the current `url` property value, with the name `temp` tacked on to the end of the filename.

3. Ask the THMetadata instance for its metadataItems. This returns an array of AVMutableMetadataItem instances containing the state of the user interface values.
4. In the export session's completion handler, you first want to determine the status of the export. If the export status is AVAssetExportSessionCompleted, which indicates the export was successful, you'll remove the old asset and replace it with the newly exported version. In this example I'm playing a bit fast and loose with the error handling for this file operation. You'll want to handle this more robustly in a production application.
5. Finally, call the private reset method, which resets the media item's prepared state and reinitializes the underlying asset.

And with that, the MetaManager app is complete! You can now run the application and make changes to the existing metadata items and save those changes back to disc.

Note

The AVAssetExportPresetPassthrough preset can be useful in certain scenarios, and it works well for the purposes of the demo app. However, be aware that it has limitations. It does enable you to modify existing metadata in an MPEG-4 or QuickTime container, but it does not enable you to add *new* metadata. The only way to add new metadata is to use one of the transcoding presets. Additionally, it cannot be used to modify ID3 tags. The framework does not support *writing* MP3 data, which is why the MP3 files in the demo app were read-only.

Summary

We've covered a lot of ground in this chapter! It was a long chapter, but arguably one of the most important as it sets the stage for the topics we'll cover from here on out. You were introduced to AVAsset and AVAssetTrack and learned the importance of retrieving their properties asynchronously using the AVAsynchronousKeyValueLoading protocol. You took a deep dive into AV Foundation's metadata functionality built around the AVMetadataItem and AVMutableMetadataItem

classes and learned the deep, dark secrets of manipulating the data they hold. Finally, you got your first introduction to `AVAssetExportSession` and put it to good use in the `MetaManager` app. This certainly isn't the last time you'll see these classes, and you'll examine their other facets as you continue along.

You're now well prepared to move forward on your AV Foundation journey!

Challenge

This chapter provided a detailed look at how to read and write some of the most common metadata, but there is certainly more to explore. Make a copy of a song or a video in your media library and fully annotate it in iTunes. Explore it with a hex editor to see how and where the data is stored. Having a good understanding of how the data is stored in the various container formats can be useful when you get into more advanced AV Foundation use cases.

4. Playing Video

Video is an incredibly powerful medium for informing, influencing, educating, and entertaining. The way we consume video has dramatically changed over the past decade thanks to video sharing sites like YouTube and Vimeo, and most recently due to devices like the iPhone and iPad. Television is quickly being usurped by streaming video; online education and video-driven virtual classrooms are becoming more commonplace, and a truly staggering number of videos are shared each and every minute on sites such as Facebook and Twitter. Clearly, there is a need for us to develop rich, dynamic video experiences to meet this growing demand, and in this chapter we dive into the details of AV Foundation's video playback capabilities. We discuss the core components involved in creating a custom video player and look at additional features the framework provides to further enhance the playback experience.

Playback Overview

A number of objects are involved when developing a custom player. In this section we'll start with a fairly high-level introduction to AV Foundation's playback capabilities by exploring the roles and relationships of the classes involved. In subsequent sections we'll dive deeper into the API details and put these classes into action by developing a custom video player. [Figure 4.1](#) provides an overview of the classes involved and their relationships.

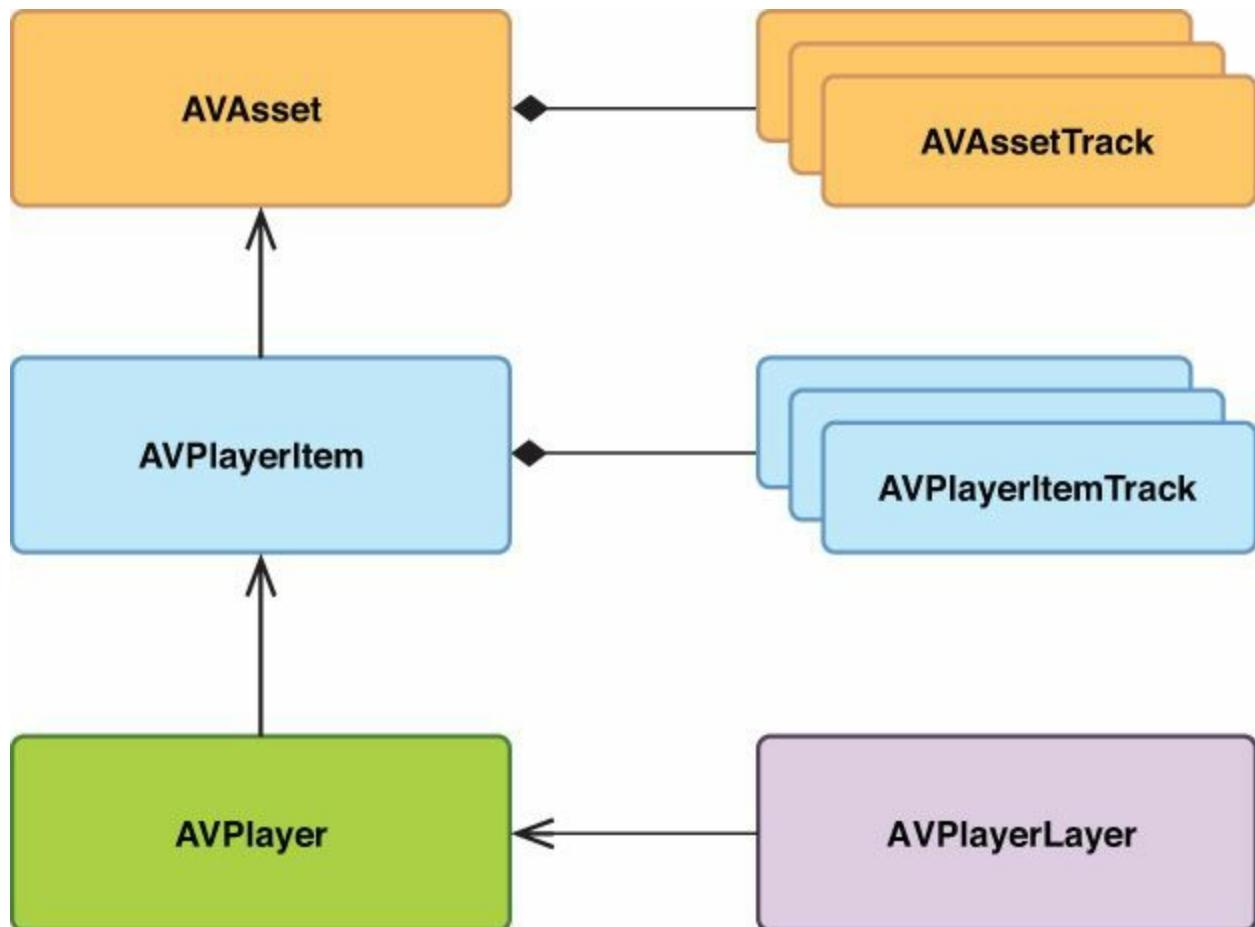


Figure 4.1 AV Foundation playback classes

AVPlayer

AV Foundation's playback support centers around a class called `AVPlayer`. An `AVPlayer` is the controller object used to play timed audio-visual media. It supports the playback of local, progressively downloaded, or streamed media delivered via the HTTP Live Streaming protocol, making it useful in a variety of playback scenarios. I should clarify that when I say "controller," I mean that in the generic sense. It's not a view or window controller, but is an object that manages the playback and timing of its associated asset. It provides the programmatic interface that you will use for developing user interfaces to control the playback of timed media.

`AVPlayer` is a nonvisual component. If your goal is to play an MP3 or an AAC audio file, this lack of a user interface wouldn't present a problem. However, this would result in a rather disappointing user experience if your goal is to play a QuickTime movie or an MPEG-4 video. To direct the video

output to a destination in the user interface, you'll make use of a class called `AVPlayerLayer`.

Note

An `AVPlayer` manages the playback of a single asset, but the framework also provides a subclass of `AVPlayer` called `AVQueuePlayer`, which can manage a queue of assets. This class can be useful when you want to play multiple items in sequence or set up an endless loop of audio or video assets.

AVPlayerLayer

`AVPlayerLayer` is built on top of Core Animation and is one of the few visual components you'll find within AV Foundation. Core Animation is the graphics rendering and animation framework available on both Mac and iOS and is responsible for the beautiful, fluid animations seen on these platforms. Core Animation is inherently time-based and, because it is built on top of OpenGL, is incredibly performant, making it a perfect fit for AV Foundation's needs.

`AVPlayerLayer` extends Core Animation's `CALayer` class and is used by the framework to render video content to the screen. This layer doesn't provide any visual controls or other adornments (those are up to you to build), but it acts as a rendering surface for the video content. An `AVPlayerLayer` is created with a pointer to an `AVPlayer` instance. This strongly binds the layer to the player, allowing it to stay in sync as changes to the player's time-based methods occur. An `AVPlayerLayer` can be used like any other `CALayer` and can be set as the backing layer for a `UIView` or `NSView` or can be manually added into an existing layer hierarchy.

`AVPlayerLayer` is a fairly simply class and is generally used as is. The one aspect of the layer you can customize is its *video gravity*. There are three different gravity values you can specify for the `videoGravity` property that determine how the video is stretched or scaled within the bounds of its containing layer. [Figures 4.2](#), [4.3](#), and [4.4](#) illustrate displaying a 16:9 video inside a 4:3 bounding rectangle so you can see the effect of the various gravity values.

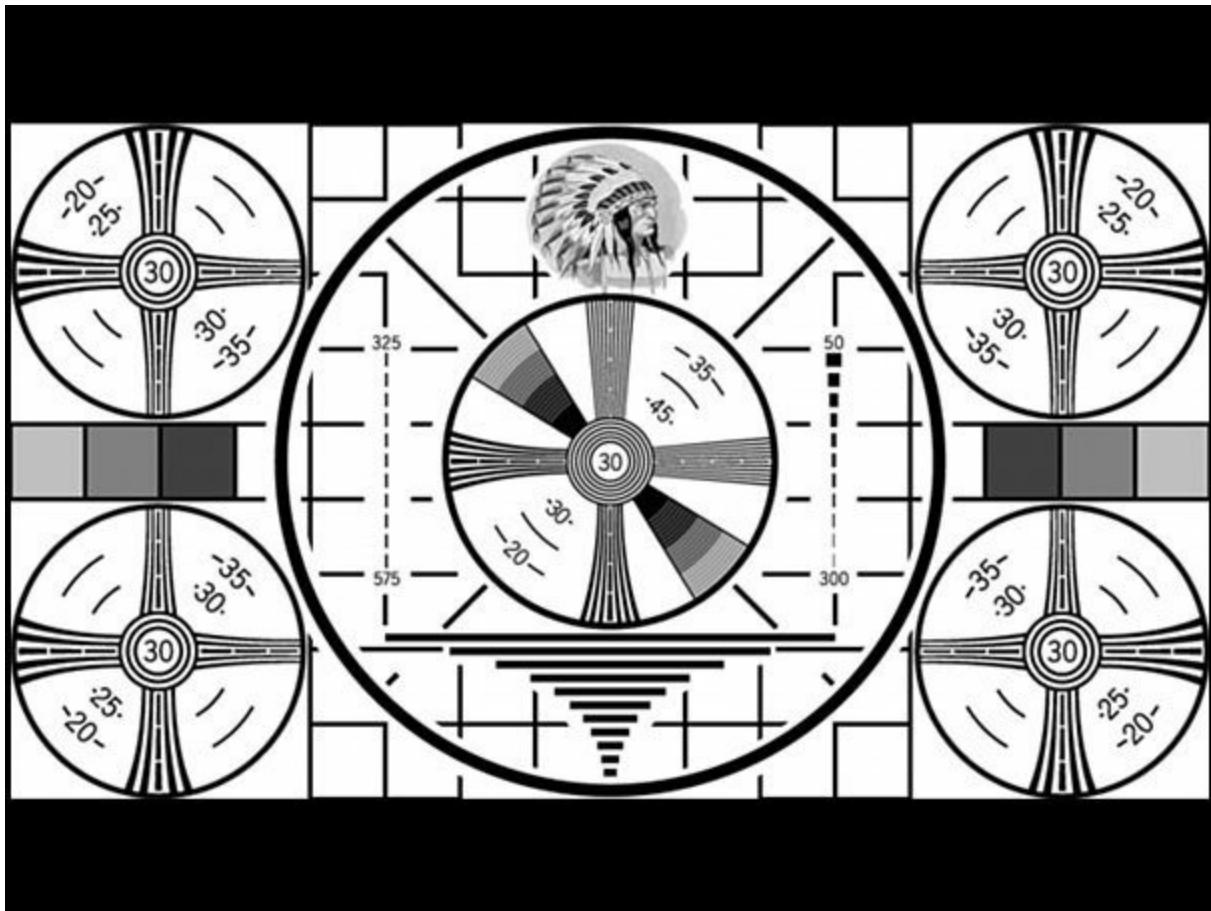


Figure 4.2 `AVLayerVideoGravityResizeAspect` will scale the video within the containing layer's bounds to preserve the video's original aspect ratio. This is the default value if not otherwise set and is appropriate in most cases.

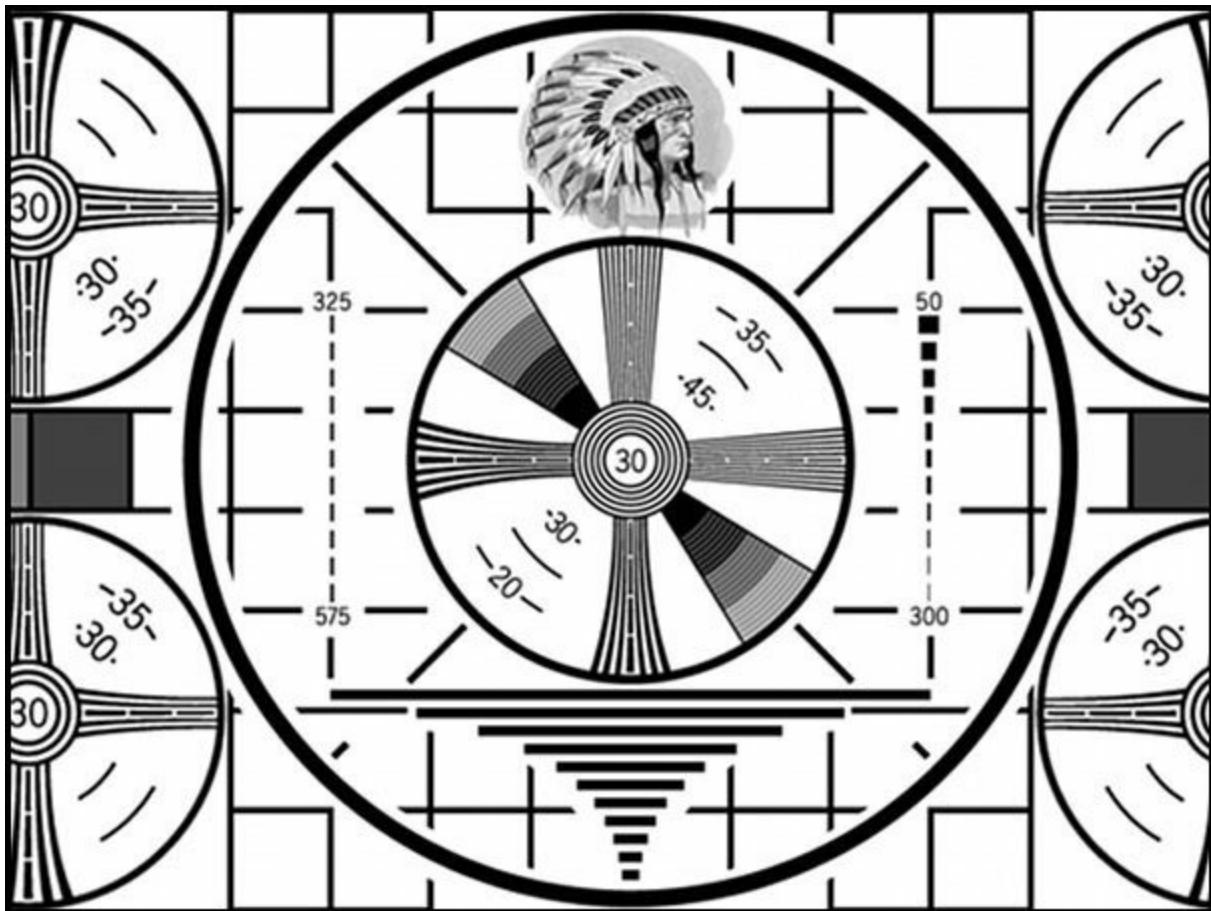


Figure 4.3 `AVLayerVideoGravityResizeAspectFill` will preserve the video's aspect ratio while scaling to fill the layer's bounds, often resulting in clipping of the video image.

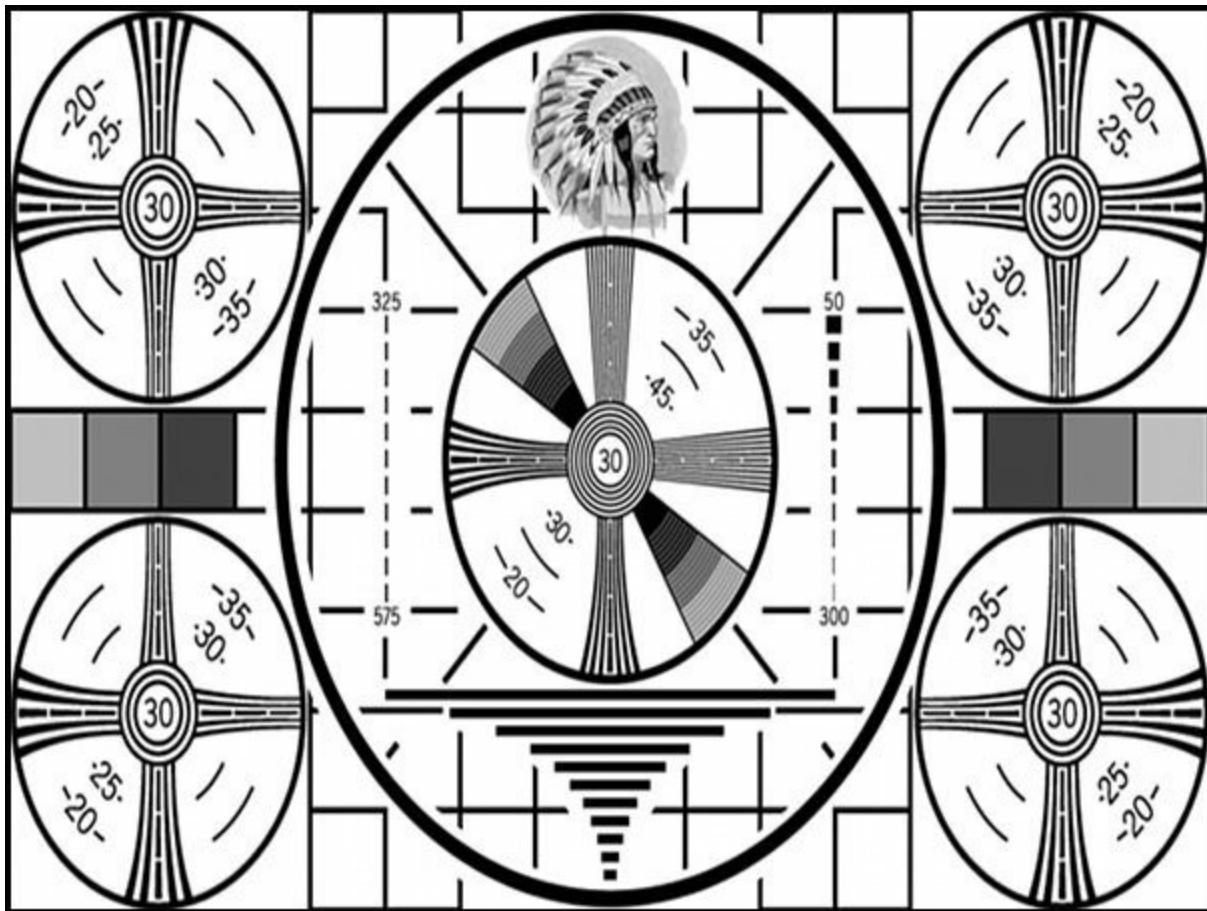


Figure 4.4 `AVLayerVideoGravityResize` will stretch the video content to match the containing layer's bounds. This is generally the least useful because it typically results in a “funhouse effect” by distorting the video image.

AVPlayerItem

Ultimately, the goal is to use `AVPlayer` to play back an `AVAsset`. If you look at the documentation for `AVAsset`, you'll find methods and properties to retrieve data, such as its creation date, metadata, and duration. However, what you won't see are methods to retrieve its current time or the ability to seek to a particular location within its media. That's because `AVAsset` models only the *static* aspects of a media resource; those persistent properties that represent the static state of the object. This means that on its own, an `AVAsset` is entirely unsuitable for playback. Whenever you want to play an asset and its associated tracks, you first need to construct their *dynamic* counterparts provided by the `AVPlayerItem` and `AVPlayerItemTrack` classes.

An `AVPlayerItem` models the dynamic perspective of the media and carries the presentation state of an asset played by an `AVPlayer`. On this class you'll find methods such as `seekToTime:` and properties to access its `currentTime` and `presentationSize`. An `AVPlayerItem` is composed of one or more tracks of media, modeled by the `AVPlayerItemTrack` class. Instances of `AVPlayerItemTrack` represent the uniformly typed media streams contained within the player item, such as its audio and video. The tracks found in an `AVPlayerItem` directly correspond to the `AVAssetTrack` instances found in the underlying `AVAsset`.

Playback Recipe

To move beyond a simple understanding of these classes, let's take a look at a small code example of how you would set up the playback stack to play a video contained within your application bundle.

[Click here to view code image](#)

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // 1. Define the asset URL
    NSURL *assetURL =
        [[NSBundle mainBundle] URLForResource:@"waves"
    withExtension:@"mp4"];

    // 2. Create an instance of AVAsset
    AVAsset *asset = [AVAsset assetWithURL:assetURL];

    // 3. Create an AVPlayerItem with a pointer to the asset to
    play
    AVPlayerItem *playerItem = [AVPlayerItem
    playerItemWithAsset:asset];

    // 4. Create an instance of AVPlayer with a pointer to the
    player item
    self.player = [AVPlayer playerWithPlayerItem:playerItem];

    // 5. Create a player layer to direct the video content
    AVPlayerLayer *playerLayer =
        [AVPlayerLayer playerLayerWithPlayer:self.player];

    // 6. Attach layer into layer hierarchy
    [self.view.layer addSublayer:playerLayer];
}
```

This example sets up the basic infrastructure required for playing back a video file. However, an additional step must be taken prior to actually playing the video content. That's because the player's associated player item is not yet in a state suitable for playback. `AVPlayerItem` provides no interface to prepare it for playback, but instead operates on a “don't call me, I'll call you” basis.

`AVPlayerItem` provides a property called `status` of type `AVPlayerItemStatus`. When first created, a player item starts with a status of `AVPlayerItemStatusUnknown`, meaning its media hasn't been loaded and has not yet been enqueued for playback. Associating an `AVPlayerItem` with an `AVPlayer` will begin enqueueing the media, but before the item is eligible to begin playback, you need to wait for its status to move from `AVPlayerItemStatusUnknown` to `AVPlayerItemStatusReadyToPlay`. The way you observe this change is by observing the `status` property via **Key-Value Observing (KVO)**.

KVO is Apple's implementation of the *Observer* pattern provided by the Foundation framework. This enables you to register an object as an observer or another object's state. When the *observed* object's state changes, the *observing* object is notified and provided the opportunity to take some action. Prior to associating an `AVPlayerItem` with an `AVPlayer`, you need to set up your code as an observer of its `status` property, as shown in the following example.

[Click here to view code image](#)

```
static const NSString *PlayerItemStatusContext;

- (void)viewDidLoad {
    ...
    AVPlayerItem *playerItem = [AVPlayerItem
        playerItemWithAsset:asset];
    [playerItem addObserver:self
        forKeyPath:@"status"
        options:0
        context:&PlayerItemStatusContext];
    self.player = [AVPlayer playerWithPlayerItem:playerItem];
}
```

```

- (void)observeValueForKeyPath:(NSString *)keyPath
    withObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {

    if (context == &PlayerItemStatusContext) {

        AVPlayerItem *playerItem = (AVPlayerItem *)object;
        if (playerItem.status == AVPlayerItemStatusReadyToPlay) {
            // proceed with playback
        }
    }
}

```

After you have observed that the player item's status has changed to `AVPlayerItemStatusReadyToPlay`, you are then free to begin playback.

Working with Time

`AVPlayer` and `AVPlayerItem` are time-based objects, but before you can make use of these features you first need to get a grasp of how time is represented in AV Foundation.

People tend to think of time in terms of days, hours, minutes, and seconds. Developers often break down time further into milliseconds and nanoseconds. As such, it might seem reasonable to represent time as a double-precision floating-point type. In fact, when we looked at `AVAudioPlayer` in [Chapter 2, “Playing and Recording Audio,”](#) you saw how it represented time as an `NSTimeInterval`, which is simply a `typedef` for a `double` value. However, representing time as a floating-point type can be problematic because floating-point operations inherently can lead to imprecisions. These imprecisions are particularly problematic when making multiple time calculations as they begin accumulating, often leading to significant timing drift, making it nearly impossible to synchronize multiple streams of media. Additionally, representing time as a floating-point type is not particularly self-describing, which makes it difficult to compare and operate on times using different timebases. AV Foundation uses a much more robust approach to representing time based on a data structure called `CMTIME`.

CMTIME

AV Foundation builds on top of a framework called Core Media. Core Media is a low-level, C-based framework that provides a lot of critical functionality to both the Mac and iOS media stacks. Although this framework generally sits behind the scenes, the one part you'll deal with frequently is its data structure, `CMTIME`. A `CMTIME` is a struct providing a rational representation of a time value—that is, a fractional value. It's defined as follows:

[Click here to view code image](#)

```
typedef struct {
    CMTimeValue value;
    CMTimeScale timescale;
    CMTimeFlags flags;
    CMTimeEpoch epoch;
} CMTime;
```

The key values from this struct are `value` and `timescale`. The `value` is a 64-bit integer and the `timescale` is a 32-bit integer, and they represent the numerator and denominator, respectively, in this rational representation of time.

Thinking of time in fractional terms can take some getting used to but becomes second nature after you've spent some time using it. Let's look at a couple examples of how to create times using the `CMTimeMake` function.

[Click here to view code image](#)

```
// 0.5 seconds
CMTime halfSecond = CMTimeMake(1, 2);

// 5 seconds
CMTime fiveSeconds = CMTimeMake(5, 1);

// One sample from a 44.1 kHz audio file
CMTime oneSample = CMTimeMake(1, 44100);

// Zero time value
CMTime zeroTime = kCMTimeZero;
```

In addition to defining `CMTime` itself, `CMTime.h` defines a large number of utility functions that simplify working with times. Like most of Apple's lower-level C frameworks, the best documentation can be found in the headers, so I'd recommend reading through `CMTime.h` to get a sense for the capabilities of the functions it defines.

Building a Video Player

In this section we'll dive into the details of AV Foundation's playback APIs by building an iOS video player (see [Figure 4.5](#)). The application will provide the capability to play both local and remote media, enabling you to play, pause, and scrub through the media timeline. After the basic functionality is created, we'll look at enhancements you can make to further improve the user experience. You'll find a starter project in the Chapter 4 directory called **VideoPlayer_Starter**.



Figure 4.5 iOS video player—ooh, unicorns!

Creating the Video View

The first step you'll take is to build the view to present the video content onscreen. In the sample project, under the `THVideoPlayer/Views` group, you'll find a class called `THPlayerView`. This class is used to display the video and provides the user interface for manipulating the video playback. Let's begin by looking at the interface for this class in [Listing 4.1](#).

Listing 4.1 `THPlayerView` Interface

[Click here to view code image](#)

```
#import "THTransport.h"
```

```
@class AVPlayer;

@interface THPlayerView : UIView
- (id)initWithPlayer:(AVPlayer *)player;
@property (nonatomic, readonly) id <THTransport> transport;
@end
```

This is a simple class with just a couple methods. It's instantiated by calling its `initWithPlayer:` initializer passing it a reference to the current `AVPlayer` instance. This enables you to direct the video output from the player to this view. The read-only `transport` property provides a handle to the visual controls displayed on this view. You'll see how this is used when we get into the implementation of the application's playback controller class. Let's move on to the implementation for this class.

The view itself is not the target of the video output; instead, we need to direct the player output to an instance of `AVPlayerLayer`. You could manually create the layer and add it into your view's layer hierarchy, but in the case of iOS, there's a more convenient way of doing so. `UIView` is always backed by a Core Animation layer. By default, this will be a generic instance of `CALayer`, but you can customize the type of backing layer used by overriding the `layerClass` method on `UIView` to return a custom `CALayer` to be used whenever the view is instantiated. This provides a more convenient way of working with `AVPlayerLayer` because it removes the need to manually create and manipulate the layer and layer hierarchy. [Listing 4.2](#) provides the implementation for the `THPlayerView` class.

Listing 4.2 THPlayerView Implementation

[Click here to view code image](#)

```
#import "THPlayerView.h"
#import "THOverlayView.h"
#import <AVFoundation/AVFoundation.h>

@interface THPlayerView ()
@property (strong, nonatomic) THOverlayView
*overlayView; // 1
@end
```

```

@implementation THPlayerView

+ (Class)layerClass
{
    return [AVPlayerLayer class]; // 2
}

- (id)initWithPlayer:(AVPlayer *)player {
    self = [super
initWithFrame:CGRectMakeZero]; // 3
    if (self) {
        self.backgroundColor = [UIColor blackColor];
        self.autoresizingMask = UIViewAutoresizingFlexibleHeight |
                               UIViewAutoresizingFlexibleWidth;

        [(AVPlayerLayer *)[self layer]
setPlayer:player]; // 4

        [[[NSBundle mainBundle]
loadNibNamed:@"THOverlayView" // 5
owner:self
options:nil];

        [self addSubview:_overlayView];
    }
    return self;
}

- (void)layoutSubviews {
    [super layoutSubviews];
    self.overlayView.frame = self.bounds;
}

- (id <THTransport>)transport {
    return self.overlayView;
}

@end

```

- 1.** Create a class extension defining a private property to store a pointer to the THOverlayView view instance. This class provides the user interface controls for manipulating the video playback.
- 2.** Override the `layerClass` class method to return an `AVPlayerLayer` class. Whenever an instance of `THPlayerView` is created, it will use `AVPlayerLayer` as its backing layer.
- 3.** Upon creation there is no inherent size, so you'll call the superclass

initializer with a zero-sized frame. It will be the responsibility of the view controller that presents this view to set its frame appropriately.

4. This is the key line of code in this class. You want to take the `AVPlayer` instance passed in the initializer and set it on the `AVPlayerLayer`. This allows the video output from the `AVPlayer` to be directed to the `AVPlayerLayer` instance.
5. The overlay view is defined in a NIB, so you'll instantiate the view by calling the `loadNibNamed:owner:options` method. With the view created and properly assigned to the `overlayView` property, you'll add it as a subview.

With the `THPlayerView` implementation complete, let's turn our attention to the `THPlayerController` class.

Creating the Video Controller

Under the `THVideoPlayer/Controllers` group, you'll find a stubbed implementation of the `THPlayerController` class. This is the class doing the bulk of the work in this application, and this is where you'll work with the core playback APIs. Let's begin by looking at the interface for this class in [Listing 4.3](#).

Listing 4.3 `THPlayerController` Interface

[Click here to view code image](#)

```
@interface THPlayerController : NSObject  
- (id)initWithURL:(NSURL *)assetURL;  
@property (strong, nonatomic, readonly) UIView *view;  
@end
```

An instance of `THPlayerController` is created by calling its `initWithURL:` initializer passing it an `NSURL` for the media to be played. `AVPlayer` can be used to play both local and streamed media, so this URL can be a local file URL or a remote HTTP URL. This class additionally provides a read-only property for its associated view so the client `UIViewController` can add the view to its view hierarchy. The view

returned is an instance of `THPlayerView`, but because those details should be hidden from clients, you'll return it as a generic `UIView`.

Switch over to the class implementation and you'll begin by creating a class extension to define the controller's internal properties (see [Listing 4.4](#)).

Listing 4.4 `THPlayerController` Class Extension

[Click here to view code image](#)

```
#import "THPlayerController.h"
#import <AVFoundation/AVFoundation.h>
#import "THTransport.h"
#import "THPlayerView.h"
#import "AVAsset+THAdditions.h"
#import "UIAlertView+THAdditions.h"

// AVPlayerItem's status property
#define STATUS_KEYPATH @"status"

// Refresh interval for timed observations of AVPlayer
#define REFRESH_INTERVAL 0.5f

// Define this constant for the key-value observation context.
static const NSString *PlayerItemStatusContext;

@interface THPlayerController () <THTransportDelegate>

@property (strong, nonatomic) AVAsset *asset;
@property (strong, nonatomic) AVPlayerItem *playerItem;
@property (strong, nonatomic) AVPlayer *player;
@property (strong, nonatomic) THPlayerView *playerView;

@property (weak, nonatomic) id <THTransport> transport;

@property (strong, nonatomic) id timeObserver;
@property (strong, nonatomic) id itemEndObserver;
@property (assign, nonatomic) float lastPlaybackRate;

@end
```

You begin the implementation of this class by first creating a class extension to define the storage properties needed by the object. You'll notice this extension adopts the `THTransportDelegate` protocol and also defines a `transport` property. There is a fair amount of communication that happens between this class and `THOverlayView`, which defines the user interface

for managing the video playback. Although these classes need to communicate, it's not necessary that they have direct knowledge of each other. To decouple this relationship, the THTransport and THTransportDelegate protocols were introduced (see [Listing 4.5](#)).

Listing 4.5 THTransport.h

[Click here to view code image](#)

```
@protocol THTransportDelegate <NSObject>

- (void)play;
- (void)pause;
- (void)stop;

- (void)scrubbingDidStart;
- (void)scrubbedToTime:(NSTimeInterval)time;
- (void)scrubbingDidEnd;

- (void)jumpedToTime:(NSTimeInterval)time;

@end

@protocol THTransport <NSObject>

@property (weak, nonatomic) id <THTransportDelegate> delegate;

- (void)setTitle:(NSString *)title;
- (void)setcurrentTime:(NSTimeInterval)time duration:
(NSTimeInterval)duration;
- (void)setScrubbingTime:(NSTimeInterval)time;
- (void)playbackComplete;

@end
```

THOverlayView adopts the THTransport protocol, which provides the formal interface for communicating with the overlay view. As changes happen in the transport, such as the user changing the scrubber position or tapping the Play/Pause button, the appropriate delegate callbacks will be made on the controller. You'll see this in action shortly, so let's move on with the implementation of THPlayerController as shown in [Listing 4.6](#).

Listing 4.6 THPlayerController Implementation

[Click here to view code image](#)

```
@implementation THPlayerController

#pragma mark - Setup

- (id)initWithURL:(NSURL *)assetURL {
    self = [super init];
    if (self) {
        _asset = [AVAsset
assetWithURL:assetURL]; // 1
        [self prepareToPlay];
    }
    return self;
}

- (void)prepareToPlay {
    NSArray *keys = @[@"tracks", @"duration", @"commonMetadata"];
    self.playerItem = [AVPlayerItem
playerItemWithAsset:self.asset // 2
automaticallyLoadedAssetKeys:keys];

    [self.playerItem
addObserver:self // 3
forKeyPath:STATUS_KEYPATH
options:0
context:&PlayerItemStatusContext];

    self.player = [AVPlayer
playerWithPlayerItem:self.playerItem]; // 4
}

self.playerView = [[THPlayerView alloc]
initWithPlayer:self.player]; // 5
self.transport = self.playerView.transport;
self.transport.delegate = self;
}

// More methods to follow ...

@end
```

- 1.** Begin by creating an AVAsset for the URL passed to the initializer. With the asset created, you'll set up the infrastructure needed to play this asset by calling the controller's `prepareToPlay` method.
- 2.** The framework automatically loads the asset's `tracks` property, saving you the need to manually load this property via the `AVAsynchronousKeyValueLoading` protocol. However, in the past, you still needed to do the

`loadValuesAsynchronouslyForKeys:completionHandle` dance for any other asset properties you needed to access. A very nice enhancement was made to `AVPlayerItem` in iOS 7 and Mac OS 10.9, enabling you to delegate the loading of an arbitrary set of properties off to the framework by creating an `AVPlayerItem` instance using its new

`initWithAsset:automaticallyLoadedAssetKeys:` or `playerItemWithAsset:automaticallyLoadedAssetKeys` initializers. Either form takes an `NSArray` as its second argument, containing the asset keys you would like to load as `AVPlayerItem` goes through its initial loading sequence. You'll make use of this approach to automatically load the `tracks`, `duration`, and `commonMetadata` properties.

3. Add `self` as an observer of the `AVPlayerItem`'s `status` property. Recall that upon creation, a player item starts out with a status of type `AVPlayerItemStatusUnknown`. The player item is not eligible to be played until its status has changed to `AVPlayerItemStatusReadyToPlay`. Key-value observing the `status` property allows you to observe this transition.
4. Create an instance of `AVPlayer` for the newly created `AVPlayerItem`. The `AVPlayer` immediately begins the process of enqueueing the media.
5. Finally, create an instance of `THPlayerView`, passing it a pointer to the `AVPlayer` instance. You'll also set up the relationship between `THPlayerController` and the `THTransport`.

Observing Status Changes

You've set up `THPlayerController` as an observer of the player item's `status` property. Before you can observe that property, you need to implement the

`observeValueForKeyPath:ofObject:change:context` method, as shown in [Listing 4.7](#).

Listing 4.7 Observing the `status` Property

[Click here to view code image](#)

```

- (void)observeValueForKeyPath:(NSString *)keyPath
    withObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {
    if (context == &PlayerItemStatusContext) {
        dispatch_async(dispatch_get_main_queue(), // 1
        ^{
            [self.playerItem removeObserver:self
            forKeyPath:STATUS_KEYPATH];
            if (self.playerItem.status ==
            AVPlayerItemStatusReadyToPlay) {
                // Set up time
                observers. // 2
                [self addPlayerItemTimeObserver];
                [self addItemEndObserverForPlayerItem];
                CMTIME duration = self.playerItem.duration;
                // Synchronize the time
                display // 3
                [self.transport
                setCurrentTime:CMTIMEGetSeconds(kCMTIMEZero)
                duration:CMTIMEGetSeconds(duration)];
                // Set the video title.
                [self.transport
                setTitle:self.asset.title]; // 4
                [self.player
                play]; // 5
            } else {
                UIAlertView *alertView =
                initWithTitle:@"Error"
                message:@"Failed to load
                video"];
            }
        });
    }
}

```

1. AV Foundation does not specify on which thread the status change notification will be made, so before you take any further action, you'll want to make sure you dispatch back to the main thread using

`dispatch_async`, passing it a reference to the main queue.

2. Set up the player's time observers by calling the private `addPlayerItemTimeObserver` and `addItemEndObserverForPlayerItem` methods. We'll discuss these methods and time observation in general in the following section.
3. Set the current time and duration on the `transport` object. This will allow the user interface's time display to be properly synchronized with the media being played. The `transport` object doesn't understand `CMTIME`, but instead deals with time in terms of seconds, represented by the `NSTimeInterval` type. You'll use the `CMTIMEGetSeconds` function to convert the `CMTIME` values into seconds. Core Media defines the constant `kCMTIMEZero` that you'll use as the starting `currentTime` argument, and you'll use the player item's duration as the second argument.
4. Pass a title string to the transport so it can present the asset's title on the display (if it exists in the asset's metadata). `AVAsset` *does not* have a `title` property. This is a category method I added to `AVAsset` to make the code more readable. This category method makes use of the metadata APIs we covered in the previous chapter and specifically, it gets the `AVMetadataCommonKeyTitle` value from the asset's `commonMetadata`. Take a look at `AVAsset+THAdditions` for details.
5. You're now ready to begin playback and do so by calling the `play` method on `AVPlayer`. Finally, as you are done observing the `status` key path, you'll want to remove `self` as the observer.

At this point you could launch the app and begin playback of one of the included videos. Although the video plays back, the user interface controls don't yet provide any functionality, and there is also no feedback provided by the user interface that time is ticking by. This brings us back to the `addPlayerItemTimeObserver` method. You need to provide an implementation for this method, but before you do we first need to talk about how to observe time changes in `AVPlayer`.

Time Observation

We discussed and have seen how to use KVO to observe the player item's

status property. KVO works well for general state observations and is useful for observing a number of properties on both `AVPlayerItem` and `AVPlayer`. Where KVO is not a good fit, however, is when it comes to observing time changes in `AVPlayer`. These types of observations are very dynamic in nature and require a finer degree of resolution than can be provided by standard key-value observations. To meet this need, `AVPlayer` provides two time-based observation methods that provide you with the ability to accurately observe times changes as they occur. Let's look at each.

Periodic Time Observation

Most commonly, you'll want to be notified of time ticking by at some regular periodic interval. This is essential when you need to update a time display or move the position of a visual playhead as time progresses. You can easily observe these types of changes by making use of `AVPlayer`'s `addPeriodicTimeObserverForInterval:queue:usingBlock:` method. This method requires that you pass it the following arguments:

- **interval**: A `CMTime` value specifying the periodic time interval at which you should be notified.
- **queue**: A *serial* dispatch queue on which the notification should be posted. Most commonly, you'll want these notifications to occur on the main queue, which is used by default if not explicitly specified. It's important to note that you *should not* use a *concurrent* dispatch queue, because the API is not written to handle concurrent queues. Doing so will result in undefined behavior.
- **block**: A callback block that will be invoked on the queue at the interval you specified. This block passes a `CMTime` value indicating the player's current time.

Boundary Time Observation

`AVPlayer` also provides a more specialized method for observing time that enables you to be notified of the traversal of various boundary points within the player's timeline. This might be useful if you would like to synchronize a user interface change or perform some nonvisual record keeping as the video plays back. For instance, you could define markers at the 25%, 50%, and 75% boundaries to determine a user's playback progress. To use this

functionality, you use the `addBoundaryTimeObserverForTimes:queue:usingBlock:` method, providing it the following arguments:

- **times**: An `NSArray` of `CMTIME` values specifying boundary points at which you'd like to be notified.
- **queue**: Like the periodic time observer, you provide this method a *serial* dispatch queue on which the notification should be posted. Specifying `NULL` is the same as explicitly setting the main queue.
- **block**: A callback block that will be invoked on the queue whenever one of your boundary points are traversed during normal playback. Oddly, the block doesn't provide the `CMTIME` value that was traversed, so you'd have to perform some additional calculations to make that determination.

The sample app doesn't make use of boundary time observations, but periodic time observations are essential to its functionality. Let's see how to make use of this periodic observation by implementing the `addPlayerItemTimeObserver` method (see [Listing 4.8](#)).

Listing 4.8 Periodic Time Observations

[Click here to view code image](#)

```
- (void)addPlayerItemTimeObserver {

    // Create 0.5 second refresh interval - REFRESH_INTERVAL == 0.5
    CMTIME interval =
        CMTIMEMakeWithSeconds(REFRESH_INTERVAL,
    NSEC_PER_SEC);                                // 1

    // Main dispatch queue
    dispatch_queue_t queue =
    dispatch_get_main_queue();                      // 2

    // Create callback block for time observer
    __weak THPlayerController *weakSelf =
    self;                                         // 3
    void (^callback)(CMTIME time) = ^(CMTIME time) {
        NSTimeInterval currentTime = CMTIMEGetSeconds(time);
        NSTimeInterval duration =
        CMTIMEGetSeconds(weakSelf.playerItem.duration);
        [weakSelf.transport setCurrentTime:currentTime
duration:duration]; // 4
```

```

    } ;

    // Add observer and store pointer for future use
    self.timeObserver
    =
        [self.player addPeriodicTimeObserverForInterval:interval
                                                queue:queue
                                              usingBlock:callback];
}


```

Note

AV Foundation uses *very* long class and method names. Coupled with blocks, the line lengths can become quite unwieldy. This method could be written more succinctly, but until the publisher decides to make its books 14 inches wider, I'll have to format the code as I have done. I would recommend writing this more succinctly in your actual project code.

1. Begin by creating a `CMTIME` to define the time interval at which you should be notified. The interval will be defined as 0.5 seconds, which will provide enough granularity to properly update the player's time display.
2. Define the dispatch queue on which the callback notifications will be posted. In almost all cases, you'll use the main queue because you typically use this type of notification to update the user interface on the main thread.
3. Define a callback block that will be invoked at the periodic interval you defined. It's critically important that the block captures a *weak* reference to `self`. Failing to do so will ensure you'll end up with hard-to-diagnose memory leaks.
4. In the body of the callback block, you'll want to convert the block's `CMTIME` value into an `NSTimeInterval` by using the `CMTIMEGetSeconds` function. Similarly, you'll convert the player item's duration as well. Passing this duration may seem redundant because you have already passed the duration to the transport in the KVO callback, but the duration can change as the media is loaded, so to

keep the user interface properly synchronized, it's best to pass it the most recent value.

5. Finally, invoke the

`addPeriodicTimeObserverForInterval:queue:usingBlock:` method with the defined arguments. This call returns an opaque `id` type pointer. You *must* maintain a *strong* reference to it for these callbacks to be made. This pointer will also be used to remove this observer.

Item End Observer

Another common event you'll likely want to observe is when the item's playback completes. This isn't a time-based observation like the ones we've looked at previously, but I tend to think of it in similar terms. When playback completes, an `AVPlayerItem` posts a notification called `AVPlayerItemDidPlayToEndTimeNotification`. The `THPlayerController` instance should register as observer of this notification so it can take the appropriate action. [Listing 4.9](#) provides the implementation of the `addItemEndObserverForPlayerItem` method.

Listing 4.9 Item End Observation

[Click here to view code image](#)

```
- (void)addItemEndObserverForPlayerItem {
    NSString *name = AVPlayerItemDidPlayToEndTimeNotification;
    NSOperationQueue *queue = [NSOperationQueue mainQueue];
    __weak THPlayerController *weakSelf =
    self; // 1
    void (^callback)(NSNotification *note) = ^(NSNotification *
    *notification) {
        [weakSelf.player
        seekToTime:kCMTimeZero // 2
            completionHandler:^(BOOL finished) {
                [weakSelf.transport
                playbackComplete]; // 3
            }];
    };
    self.itemEndObserver
    = // 4
```

```

    [[NSNotificationCenter defaultCenter]
addObserverForName:name
                                         object:self
                                         queue:queue
usingBlock:^(NSNotification *note)
{
    self.itemEndObserver = nil;
}
- (void)dealloc {
    if (self.itemEndObserver)
    {
        // 5
        NSNotificationCenter *nc = [NSNotificationCenter
defaultCenter];
        [nc removeObserver:self.itemEndObserver
name:AVPlayerItemDidPlayToEndTimeNotification
object:self.player.currentItem];
        self.itemEndObserver = nil;
    }
}

```

1. Prior to defining the block, you'll first define a *weak* reference to `self`. Similar to the callback block used for the periodic time observation, failing to capture a weak reference to `self` will result in memory leaks. These block-based retain cycles are difficult to diagnose and often result in rapidly thinning hair.
2. When the playback completes, you'll reposition the playhead cursor back to the zero position by calling `seekToTime:kCMTimeZero` on the player instance.
3. When the seek call in #2 completes, you'll notify the transport that the playback is complete so it can reset its time display and scrubber.
4. You add the `itemEndObserver` as the observer of that notification by registering with `NSNotificationCenter`, providing it the arguments you defined.
5. Finally, you'll override the `dealloc` method to remove the `itemEndObserver` as an observer when the controller is deallocated.

Run the application. You can see that the video plays and, as time ticks by, you'll see the current and remaining time labels update their values, and you'll see the time scrubber update its playhead position accordingly.

Let's move on to implement the various delegate callback methods so the

transport controls work as expected.

Transport Delegate Callbacks

Let's begin with the implementation of the simple transport callbacks provided by the `THTransportDelegate` protocol. [Listing 4.10](#) provides the implementation of these methods.

Listing 4.10 Transport Delegate Callbacks

[Click here to view code image](#)

```
- (void)play {
    [self.player play];
}

- (void)pause {
    self.lastPlaybackRate = self.player.rate;
    [self.player pause];
}

- (void)stop {
    [self.player setRate:0.0f];
    [self.transport playbackComplete];
}

- (void)jumpedToTime:(NSTimeInterval)time {
    [self.player seekToTime:CMTIMEMakeWithSeconds(time,
NSEC_PER_SEC)];
}
```

The implementations of the `play` should be self-explanatory because it delegates to the player's method of the same name. Likewise, the `pause` method delegates the player's `pause` method, but also captures the `lastPlaybackRate` for housekeeping purposes. The `stop` method calls `setRate:` passing it a value of 0, which is equivalent to calling `pause`, but demonstrates an alternate way of achieving the same effect. It also calls `playbackComplete` on the transport so it can update the position of its scrubber display. The `jumpedToTime:` method makes use of the player's `seekToTime:` method to jump to an arbitrary point within the timeline. You'll see how this method is used later in the chapter.

Next, let's take a look at how to implement the scrubbing-related methods.

There are three methods to implement, reflecting the three event phases a user goes through while interacting with the `UISlider` control (see [Listing 4.11](#)).

Listing 4.11 Scrubbing Methods

[Click here to view code image](#)

```
- (void)scrubbingDidStart // 1
{
    self.lastPlaybackRate = self.player.rate;
    [self.player pause];
    [self.player removeTimeObserver:self.timeObserver];
}

- (void)scrubbedToTime:(NSTimeInterval)time // 2
{
    [self.playerItem cancelPendingSeeks];
    [self.player seekToTime:CMTIMEMakeWithSeconds(time,
NSEC_PER_SEC)];
}

- (void)scrubbingDidEnd // 3
{
    [self addPlayerItemTimeObserver];
    if (self.lastPlaybackRate > 0.0f) {
        [self.player play];
    }
}
```

1. `scrubbingDidStart` is called in response to a touch down event (`UIControlEventTouchDown`). In this method you'll capture the current playback rate and then pause the player. You'll capture the current rate so it can be restored when the scrubbing ends. Additionally, you'll remove the current periodic time observer, because you don't want these events to fire as the user is scrubbing through the media.
2. `scrubbedToTime` is called in response to a value changed event (`UIControlEventValueChanged`) on the `UISlider` instance. Because this method will be called rapidly as the user moves the slider position, you should begin by first calling `cancelPendingSeeks` on the player item. This is a performance optimization to prevent seek operations from piling up if the previous seek request hasn't completed.

You'll begin a new seek by calling `seekToTime:` translating the `NSTimeInterval` value into a `CMTIME`.

3. `scrubbingDidEnd` is called in response to a touch-up inside event (`UIControlEventTouchUpInside`), indicating the user has completed the scrubbing operation. In this method you'll re-add the periodic time observer by calling `addPlayerItemTimeObserver`. You'll then check the `lastPlaybackRate` value, and if it was greater than 0, which indicates the video was playing back, you'll begin playing the video again.

With that, the core video playback behavior is complete! Run the application and you should now be able to play, pause, and scrub through the video. With the core behavior working, it's time to look at some ways you could further improve the player by adding some features and functionality to enhance the video playback experience.

Creating a Visual Scrubber

You may have noticed the button in the upper-right corner of the player with the label Show. If you tapped it, you noticed it toggles the display of a black bar below the main navigation bar. Not a particularly useful feature as it stands, but let's see if we can put this real estate to better use.

A useful utility class you'll find in AV Foundation is called `AVAssetImageGenerator`. This class can be used to extract images from an `AVAsset`'s video tracks. This enables you to generate one or more thumbnails that could be used to enhance your application's user interface.

`AVAssetImageGenerator` provides two methods to retrieve images from a video asset:

- `copyCGImageAtTime:actualTime:error:` enables you to capture an image at a specified time. This is most useful if you want to capture a single image, perhaps to display as a video thumbnail in a list of videos.
 - `generateCGImagesAsynchronouslyForTimes:completion:` enables you to generate a sequence of images for the times specified in the first argument. This provides you with a performant way of generating a collection of images in a single call.
-

Note

AVAssetImageGenerator can be used to generate images for both local and progressively downloaded assets. It can't, however, generate images from an HTTP Live Stream.

One nice feature you could build using this functionality is to create a visual scrubber. Instead of showing the standard scrubber bar in the bottom toolbar, you'll build a visual scrubber so a user can more easily identify a location within the timeline and immediately jump to it. Let's take a look at how to build this functionality (see [Listing 4.12](#)).

Listing 4.12 Image Generation

[Click here to view code image](#)

```
#import "THPlayerController.h"
#import <AVFoundation/AVFoundation.h>
#import "THTransport.h"
#import "THPlayerView.h"
#import "AVAsset+THAdditions.h"
#import "UIAlertView+THAdditions.h"
#import "THThumbnail.h"

...
@interface THPlayerController () <THTransportDelegate>

@property (strong, nonatomic) AVAsset *asset;
@property (strong, nonatomic) AVPlayerItem *playerItem;
@property (strong, nonatomic) AVPlayer *player;
@property (strong, nonatomic) THPlayerView *playerView;

@property (weak, nonatomic) id <THTransport> transport;

@property (strong, nonatomic) id timeObserver;
@property (strong, nonatomic) id itemEndObserver;
@property (assign, nonatomic) float lastPlaybackRate;

@property (strong, nonatomic) AVAssetImageGenerator
*imageGenerator;

@end
```

You'll import the THThumbnail.h header. The THThumbnail class is a

simple model object in the project you'll use to store the captured image and its associated time. You'll also add a new property of type AVAssetImageGenerator.

Next, add a new method called generateThumbnails and call this method from inside your status observation callback method, as shown in [Listing 4.13](#).

Listing 4.13 Invoking generateThumbnails

[Click here to view code image](#)

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {
    if (context == &PlayerItemStatusContext) {
        dispatch_async(dispatch_get_main_queue(), ^{
            [self.playerItem removeObserver:self
forKeyPath:STATUS_KEYPATH];
            if (self.playerItem.status ==
AVPlayerItemStatusReadyToPlay) {
                // Set up time observers.
                [self addPlayerItemTimeObserver];
                [self addItemEndObserverForPlayerItem];
                CMTIME duration = self.playerItem.duration;
                // Synchronize the time display
                [self.transport
setCurrentTime:CMTIMEGetSeconds(kCMTIMEZero)
duration:CMTIMEGetSeconds(du:
                // Set the video title.
                [self.transport setTitle:self.asset.title];
                [self.player play];
[self generateThumbnails];
            } else {
                UIAlertView *alertView =
UIAlertView *alertView initWithTitle:@"Error"
message:@"Failed to load
            }
        }
    }
}
```

```

video"] ;
        }
    } );
}
}

- (void)generateThumbnails {
}

```

With the infrastructure in place, let's implement this method. [Listing 4.14](#) provides implementation for the generateThumbnails method.

Listing 4.14 **generateThumbnails** Implementation

[Click here to view code image](#)

```

- (void)generateThumbnails {

    self.imageGenerator
=                               // 1
    [AVAssetImageGenerator
assetImageGeneratorWithAsset:self.asset];

    // Generate the @2x equivalent
    self.imageGenerator.maximumSize = CGSizeMake(200.0f,
0.0f);                      // 2

    CMTIME duration = self.asset.duration;

    NSMutableArray *times = [NSMutableArray
array];                         // 3
    CMTIMEValue increment = duration.value / 20;
    CMTIMEValue currentValue = kCMTimeZero;
    while (currentValue <= duration.value) {
        CMTIME time = CMTimeMake(currentValue,
duration.timescale);
        [times addObject:[NSValue valueWithCMTime:time]];
        currentValue += increment;
    }

    __blockNSUInteger imageCount =
times.count;                     // 4
    __block NSMutableArray *images = [NSMutableArray array];

    AVAssetImageGeneratorCompletionHandler
handler;                         // 5

    handler = ^ (CMTIME requestedTime,

```

```

        CGImageRef imageRef,
        CMTime actualTime,
        AVAssetImageGeneratorResult result,
        NSError *error) {

    if (result == AVAssetImageGeneratorSucceeded)
        // 6
        UIImage *image = [UIImage imageWithCGImage:imageRef];
        id thumbnail =
            [THThumbnail thumbnailWithImage:image
time:actualTime];
        [images addObject:thumbnail];
    } else {
        NSLog(@"Failed to create thumbnail image.");
    }

    // If the decremented image count is at 0, we're all done.
    if (--imageCount == 0)
    {
        // 7
        dispatch_async(dispatch_get_main_queue(), ^{
            NSString *name =
THThumbnailsGeneratedNotification;
            NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
            [nc postNotificationName:name object:images];
        });
    }
};

[self.imageGenerator
generateCGImagesAsynchronouslyForTimes:times           // 8
completionHandler:^(
}

```

- 1.** Begin by creating a new `AVAssetImageGenerator` instance by passing it a reference to the controller's `asset` property. It's critical that you maintain a strong reference to this object. Failing to pay attention to this point will leave you frustrated, because your callbacks will never be invoked.
- 2.** `AVAssetImageGenerator` provides a few properties you can use to configure the image generation. Although it provides sensible defaults for most properties, the one you should almost always explicitly configure is its `maximumSize`. By default, images will be captured at their native dimension. If you're working with a 720p or

1080p video, this results in very large images being created. Setting the `maximumSize` property automatically scales the image to the size you need and will *dramatically* improve performance. You'll specify a `CGSize` with a width of 200 and a height of 0. This will ensure the images are generated at the specified width and will automatically size the height according to the aspect ratio of the video.

3. The next thing to do is perform some calculations to generate a collection of `CMTime` values specifying the locations in the video to capture. The code generates 20 `CMTime` values evenly spaced over the video's timeline. It loops over the duration of the video, creating a new time using the `CMTimeMake` function, and then wraps the resulting `CMTime` as an `NSValue` so it can be stored in the `times` array.
4. Define a `_block` variable called `imageCount` based on the number of elements in the `times` array. This will be used to determine when all images have been processed. Additionally, define another `_block` variable, this time of type `NSMutableArray` called `images`. This will be used to store the collection of generated images. The `_block` qualifier is used to ensure that the callback block operates on these pointers directly and not on a copy.
5. Next you'll define a callback block of type `AVAssetImageGeneratorCompletionHandler`. This is one of the longer block definitions you'll run across, so let's take a look at its arguments:
 - **requestedTime**: The original time you requested. This corresponds to a value in the `times` array you specified in the call to generate the images.
 - **imageRef**: The `CGImageRef` that was generated or `NULL` if an image for the given time couldn't be generated.
 - **actualTime**: The time the image was *actually* generated. Based on certain efficiencies, this can be different from the time you requested. You can tailor how closely the `requestedTime` and `actualTime` match by setting the `requestedTimeToleranceBefore` and `requestedTimeToleranceAfter` values on the `AVAssetImageGenerator` instance prior to image generation.

- **result**: An `AVAssetImageGeneratorResult` indicating whether the image generation was successful, failed, or was cancelled.
- **error**: An `NSError` pointer you can interrogate if you received an `AVAssetImageGeneratorResult` of `AVAssetImageGeneratorFailed`.

6. If the result value came back as

`AVAssetImageGeneratorSucceeded`, indicating the image was successfully generated, create a new `UIImage` based on the returned `CGImageRef`. Then create a new `THThumbnail` instance wrapping the image and time and add it to the array.

7. On each invocation of the callback block, you'll decrement the `imageCount` property and determine if it's 0, which indicates all images have been processed. If so, you'll post a new, application-specific notification called

`THThumbnailsGeneratedNotification` passing the collection of images as its `object` argument. This notification will be consumed by the view tier and used to generate the visual scrubber.

Run the application again. Now when you tap the Show button you'll see that the black bar has been replaced with a nice collection of thumbnails indicating the various points in time within the video file. Tapping on an image invokes the delegate's `jumpedToTime:` method that you implemented earlier. You'll notice that `AVAssetImageGenerator` was able to generate images for both the local and remote asset, but as is to be expected, it takes longer to generate the images for the remote resource. Some greater efficiencies could be made in this case to improve the user experience, such as building the visual layout as each image is returned, or writing to an image cache and having the view poll this cache at some regular interval.

Note

Most playback use cases can be tested in the iOS Simulator. However, you'll generally find the performance is much better if you test on an actual device.

Showing Subtitles

It's important that we make our applications accessible to as many people as possible. This means we should make our applications available to users in their native language as well consider the needs of those with hearing impairment or other accessibility needs. One way you can improve the user experience for users of your video player application is to display subtitles whenever available. AV Foundation provides robust support for displaying subtitles or closed captions. `AVPlayerLayer` automatically provides support for rendering these elements, and it's simply up to you to tell it what to render. The way you do that is through the use of the `AVMediaSelectionGroup` and `AVMediaSelectionOption` classes.

An `AVMediaSelectionOption` represents an alternate media presentation within an `AVAsset`. An asset may contain alternate media presentations such as alternate audio, video, or text tracks. These tracks could be language-specific audio tracks, alternate camera angles, or as we're interested in at the moment, language-specific subtitles. The way you can determine which alternate tracks exist is through a property on `AVAsset` called

`availableMediaCharacteristicsWithMediaSelectionOption` (I mentioned the AV Foundation team has a penchant for really long names, remember?) This property returns an array of strings indicating the media characteristics of the available options contained within the asset.

Specifically, it returns an array of one or more of the following string values: `AVMediaCharacteristicVisual` (video), `AVMediaCharacteristicAudible` (audio), or `AVMediaCharacteristicLegible` (subtitles or closed captions).

After you've asked for the available media characteristics, you can call `AVAssets mediaSelectionGroupForMediaCharacteristic:` method, passing it the particular media characteristic for the options you'd like to retrieve. This method returns an `AVMediaSelectionGroup`, which acts as a container for one or more mutually exclusive instances of `AVMediaSelectionOption`. Let's look at a simple example:

[Click here to view code image](#)

```
NSArray *mediaCharacteristics =
    self.asset.availableMediaCharacteristicsWithMediaSelectionOption;
for (NSString *characteristic in mediaCharacteristics) {
```

```

AVMediaSelectionGroup *group =
    [self.asset
mediaSelectionGroupForMediaCharacteristic:characteristic];
    NSLog(@"%@", characteristic);
    for (AVMediaSelectionOption *option in group.options) {
        NSLog(@"Option: %@", option.displayName);
    }
}
}

```

Running this code for an asset containing one or more subtitles would produce output similar to the following:

[Click here to view code image](#)

```

[AVMediaCharacteristicLegible]
Option: English
Option: Italian
Option: Portuguese
Option: Russian
[AVMediaCharacteristicAudible]
Option: English

```

In this example you can see that multiple subtitle tracks exist along with a single English audio track.

After you've loaded the appropriate AVMediaSelectionGroup and identified the desired AVMediaSelectionOption, the next step is to put it into action. This is done by calling

`selectMediaOption:inMediaSelectionGroup:` on the active AVPlayerItem. For instance, if you wanted to display the Russian subtitle, you could write the following:

[Click here to view code image](#)

```

AVMediaSelectionGroup *group =
    [self.asset
mediaSelectionGroupForMediaCharacteristic:characteristic];
NSLocale *russianLocale = [[NSLocale alloc]
initWithLocaleIdentifier:@"ru_RU"];
NSArray *options =
    [AVMediaSelectionGroup
mediaSelectionOptionsFromArray:group.options
                                withLocale:russianL
AVMediaSelectionOption *option = [options firstObject];
[self.playerItem selectMediaOption:option
inMediaSelectionGroup:group];

```

Let's put this into action in the Video Player app by adding a couple new methods to the THPlayerController class. Begin by making the

following changes as shown in [Listing 4.15](#).

Listing 4.15 **loadMediaOptions** Set Up

[Click here to view code image](#)

```
- (void)prepareToPlay {
    NSArray *keys = @[
        @"tracks",
        @"duration",
        @"commonMetadata",
        @"availableMediaCharacteristicsWithMediaSelectionOptions"
    ];
    ...
}

- (void)observeValueForKeyPath:(NSString *)keyPath
    withObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {

    if (context == &PlayerItemStatusContext) {
        dispatch_async(dispatch_get_main_queue(), ^{
            if (self.playerItem.status ==
                AVPlayerItemStatusReadyToPlay) {
                ...
                [self loadMediaOptions];
            } else {
                UIAlertView *alertView =
                    [[UIAlertView alloc] initWithTitle:@"Error"
                                               message:@"Failed to load
video"];
            }
        });
    }
}

- (void)loadMediaOptions {
```

In the `prepareToPlay` method, you'll want to add the `availableMediaCharacteristicsWithMediaSelectionOptio`

property to its list of properties to automatically load. It's necessary to load this property prior to calling any of the media selection APIs in order to prevent blocking the main thread. You'll invoke the `loadMediaOptions` method after the player item is ready to play.

The implementation of the `loadMediaOptions` method is shown in [Listing 4.16](#).

Listing 4.16 `loadMediaOptions` Implementation

[Click here to view code image](#)

```
- (void)loadMediaOptions {
    NSString *mc =
    AVMediaCharacteristicLegible; // 1
    AVMediaSelectionGroup *group =
        [self.asset
    mediaSelectionGroupForMediaCharacteristic:mc]; // 2
    if (group) {
        NSMutableArray *subtitles = [NSMutableArray
array]; // 3
        for (AVMediaSelectionOption *option in group.options) {
            [subtitles addObject:option.displayName];
        }
        [self.transport
setSubtitles:subtitles]; // 4
    } else {
        [self.transport setSubtitles:nil];
    }
}
```

1. You're interested in finding only the subtitle options present in the asset so define a media characteristic string with the value `AVMediaCharacteristicLegible`.
2. Ask for the `AVMediaSelectionGroup` corresponding to the defined media characteristic.
3. Assuming a group was found (the application's local asset contains subtitles, the remote asset does not), create an array of user-presentable strings to be passed to the view tier by asking each option for its `displayName` property.
4. Finally, set the collection of subtitle strings on the transport so they can be presented in its subtitle selection interface. In the `else` condition,

pass `nil`, indicating that no interface should be presented.

When the user selects a subtitle, you'll need a method to handle that selection and activate the corresponding `AVMediaSelectionOption` on the current player item. [Listing 4.17](#) shows how to implement this method.

Listing 4.17 Handling Subtitle Selection

[Click here to view code image](#)

```
- (void)subtitleSelected:(NSString *)subtitle {
    NSString *mc = AVMediaCharacteristicLegible;
    AVMediaSelectionGroup *group =
        [self.asset
mediaSelectionGroupForMediaCharacteristic:mc]; // 1
    BOOL selected = NO;
    for (AVMediaSelectionOption *option in group.options) {
        if ([option.displayName isEqualToString:subtitle]) {
            [self.playerItem
selectMediaOption:option // 2
                inMediaSelectionGroup:group];
            selected = YES;
        }
    }
    if (!selected) {
        [self.playerItem
selectMediaOption:nil // 3
                inMediaSelectionGroup:group];
    }
}
```

1. Retrieve the `AVMediaSelectionGroup` for the legible options contained in the asset.
2. Loop through all the group's options and find the `AVMediaSelectionOption` matching the subtitle string passed to `subtitleSelected:` method. After the correct option has been found, you'll activate it by calling `selectMediaOption:inMediaSelectionGroup:` on the player item. This immediately enables the display of the selected subtitle on the `AVPlayerLayer`.
3. If the user selected the “None” option from the subtitle selection list, set a `nil` value for the selected media option in order to remove the

subtitles from display.

The one last thing you need to do is open up the `VideoPlayer-Prefix.pch` file and change the `ENABLE_SUBTITLES` define from 0 to 1. This enables the transport view to display the appropriate subtitle selection interface if subtitles are present in the current media.

Run the application again. In the bottom-right corner of the transport bar you'll see a new button. Select it to see the available subtitles. Make a selection and voilà! Subtitle magic.

Airplay

The one last enhancement we'll discuss is incorporating AirPlay functionality into the Video Player app. AirPlay is Apple's technology enabling users to wirelessly stream audio and video content to an Apple TV or audio-only content to a variety of third-party audio systems. If you're an owner of an Apple TV or one of these other audio systems, you know how incredibly useful this feature can be. The good news is it's really easy to incorporate this functionality into your apps.

`AVPlayer` has a property called `allowsExternalPlayback`, providing you the ability to enable or disable AirPlay playback. The default value of this property is YES, meaning your playback app automatically supports this behavior without any additional effort from you. Although this is generally the desired behavior, if you had a compelling reason why AirPlay should be disabled, you can set this value to NO.

Providing a Route Picker

iOS provides a global interface for selecting an AirPlay route. The user interface and gestures for displaying the interface differ depending on your iOS version. On iOS 6 and earlier, you would double-tap the Home button to bring up the dock, and swipe to the right until you get to the AirPlay route chooser, as shown in [Figure 4.6](#).

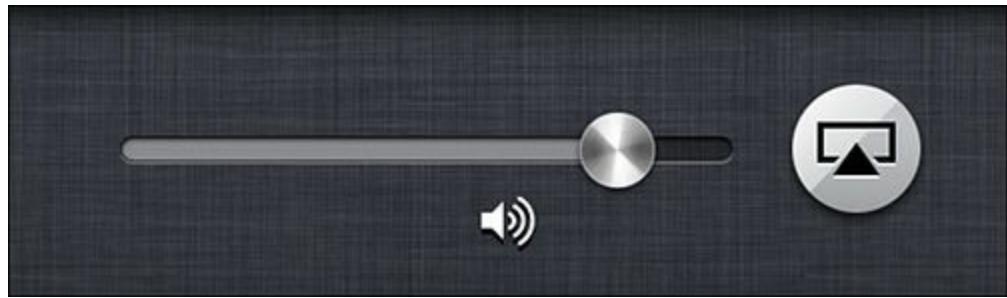


Figure 4.6 iOS 6 AirPlay route button

iOS 7 and later provides a much more convenient way of accessing this interface. Swipe up from the bottom of the screen to bring up Control Center, and select the AirPlay button, as shown in [Figure 4.7](#).

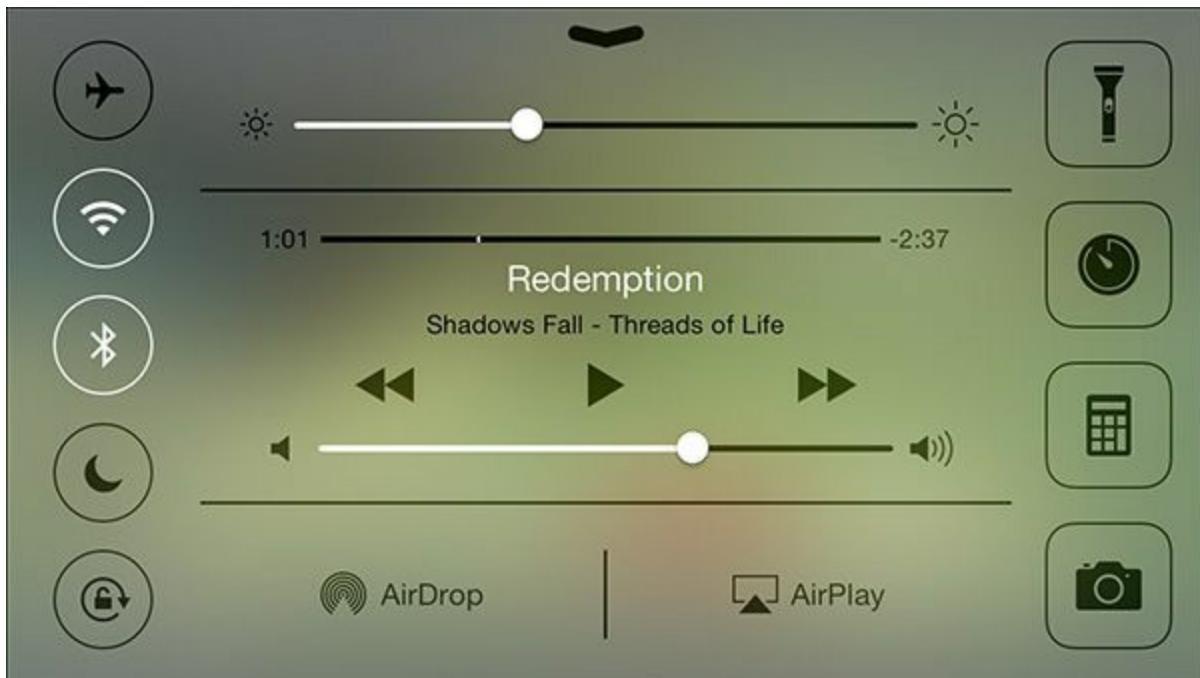


Figure 4.7 iOS 7 AirPlay route button

Although using the global route chooser helps the user achieve the desired result, it's far from an ideal user experience. On iOS 6 in particular, it takes the user out of your app, which disrupts the user's workflow. It's also important to note that many users may not be aware of this global interface and may completely overlook this incredibly useful feature. Instead, you should provide the AirPlay route selection interface obviously and conveniently inside your app. Interestingly, there is no AirPlay framework or API for you to use; instead, you make use of a class in the Media Player framework called `MPVolumeView`.

To use this component, you link to and import the MediaPlayer framework (<MediaPlayer/MediaPlayer.h>) and create an instance of an MPVolumeView as shown in the following example.

[Click here to view code image](#)

```
CGRect rect = // desired frame
MPVolumeView *volumeView = [[MPVolumeView alloc]
initWithFrame:rect];
[self.view addSubview:volumeView];
```

A default instance of MPVolumeView provides up to two user interface elements. As the name suggests, it displays a volume slider enabling the user to control the overall system volume. This provides the onscreen equivalent of pressing the hardware volume controls on the side of an iOS device. If the user has an AirPlay-enabled device on his or her network, it will additionally show an AirPlay route selector button. Tapping this button presents a listing of all available AirPlay routes.

If you're interested only in displaying the route button, you can make the following modifications:

[Click here to view code image](#)

```
MPVolumeView *volumeView = [[MPVolumeView alloc] init];
volumeView.showsVolumeSlider = NO;
[volumeView sizeToFit];
[transportView addSubview:volumeView];
```

It's important to understand that the route selector button will be shown *only* if the user has a valid AirPlay destination and is Wi-Fi enabled. Unless both of those conditions are true, MPVolumeView will automatically hide the button.

Note

MPVolumeView will be displayed only if running on an iOS device. It will not be displayed while running in the iOS Simulator.

I won't have you go through the process of implementing this in the app, because it's no more complex than what was shown in the previous examples. The app is already built to show the route button if possible, but you do need to open the VideoPlayer-Prefix.pch and change the ENABLE_AIRPLAY define from 0 to 1. If you have an Apple TV or an

AirPlay-enabled audio system, you'll see the AirPlay route selector when you run the application.

For more information on AirPlay and ways you can further take advantage of this amazing technology, be sure to check out the AirPlay Overview¹ document available from the Apple Developer Center.

1.

<https://developer.apple.com/library/ios/documentation/AVFoundation/Conceptual/AirPlayGuide/Introduction.html>

Summary

This chapter provided an in-depth look at AV Foundation's video playback capabilities. You should now have a good understanding of how to play instances of `AVPlayerItem` via an `AVPlayer` and direct the video output to an instance of `AVPlayerLayer`. You also got your first look at `AVAssetImageGenerator`, which was put to good use building the player's visual scrubber. You'll find it to be a very useful class in a variety of scenarios using AV Foundation. Finally, you saw ways you can enhance the video playback experience by incorporating AirPlay and using `AVMediaSelectionGroup` and `AVMediaSelectionOption` to display subtitles. The sample application you built in this chapter should provide a good jumping off point for developing any video playback solution you need.

Challenge

I'll present you with two challenges in this chapter:

1. We've covered some of the most important aspects of `AVPlayer`, `AVPlayerItem`, and `AVPlayerLayer`, but there are additional interesting features to explore. There are additional seek operations to try, additional properties to interrogate, and logging information to retrieve. The sample application will provide you with a great test harness to experiment with these additional features.
2. Take the time to familiarize yourself with `CMTIME`. It plays an important role in many of AV Foundation's features, so you'll want to be well versed in its use. Open the `CMTIME.h` header to discover the functions it provides. This header also provides the best source of documentation on its various structures, macros, and functions. I would recommend creating a simple command-line application that links

against the Core Media framework and spend time experimenting. Learn to create, compare, add, and subtract `CMTime` values. I assure you this will be time well spent.

5. Using AV Kit

In [Chapter 4](#), “[Playing Video](#),” you were presented with a detailed look at using `AVPlayer` and `AVPlayerItem` to build a custom video player. Building a custom video player is desirable in many cases because it provides you complete control over the player’s behavior and its user interface. However, what if you wanted to be able to leverage the power of AV Foundation, but provide a user interface and experience that matches the Videos app on iOS or QuickTime Player on Mac OS X? It would require a fair amount of work to match the look and feel of those players and would also require you to maintain multiple user interfaces to match the look of each iOS or Mac OS X version. [*Infomercial-guy voice*] There has to be a better way! Fortunately, there is, thanks to the AV Kit framework.

AV Kit simplifies the process of building AV Foundation-based video players that match the look and feel of the default operating system players. This framework was first introduced in Mac OS X Mavericks, and starting with iOS 8, is now available on iOS as well. This chapter primarily focuses on AV Kit for OS X because it provides more capabilities than its iOS counterpart, but let’s begin with a quick look at the iOS version of AV Kit.

AV Kit for iOS

The iOS Media Player framework has long provided the `MPMoviePlayerController` and `MPMoviePlayerViewController` classes that provide a simple way of incorporating full-featured video playback into your apps. `MPMoviePlayerController` provides standard playback controls, can be embedded as a subview or presented full screen, supports streaming audio and video content to an Apple TV via AirPlay, and offers a number of other useful features. With all this functionality, why would we need anything else? The principle drawback of `MPMoviePlayerController` is that it’s very much a black box component. It’s built on top of AV Foundation, but unfortunately hides those underpinnings. This prevents you from taking advantage of some of the more advanced features offered by `AVPlayer` and `AVPlayerItem`. Starting with iOS 8, you now have a more powerful alternative provided by the AV Kit framework.

AV Kit for iOS is fairly modest by framework standards—containing only a single class called `AVPlayerViewController`. This is a `UIViewController` subclass used to display and control the playback of an `AVPlayer` instance. `AVPlayerViewController` has a pretty minimal interface providing the following properties:

- **player**: The `AVPlayer` instance used to play the media content.
- **showsPlaybackControls**: A Boolean value indicating whether to show or hide the playback controls.
- **videoGravity**: An `NSString` used to set the video gravity of its internal `AVPlayerLayer` instance. See [Chapter 4](#) if you need a refresher on the video gravities supported by `AVPlayerLayer`.
- **readyForDisplay**: A Boolean value you can observe to determine whether the video content has been primed and is ready for display.

Despite its unassuming interface, this class provides a lot of utility value. To illustrate this point, let's look at a simple example of this class in action.

[Click here to view code image](#)

```
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    NSURL *url =
        [[NSBundle mainBundle] URLForResource:@"video"
withExtension:@"m4v"];

    AVPlayerViewController *controller = [[AVPlayerViewController
alloc] init];
    controller.player = [AVPlayer playerWithURL:url];

    self.window.rootViewController = controller;

    return YES;
}

@end
```

This example creates a new `AVPlayerViewController`, sets an instance of `AVPlayer` on it, and sets the controller as the window's `rootViewController`. Although very little code was written, running this example produces the player shown in [Figure 5.1](#).

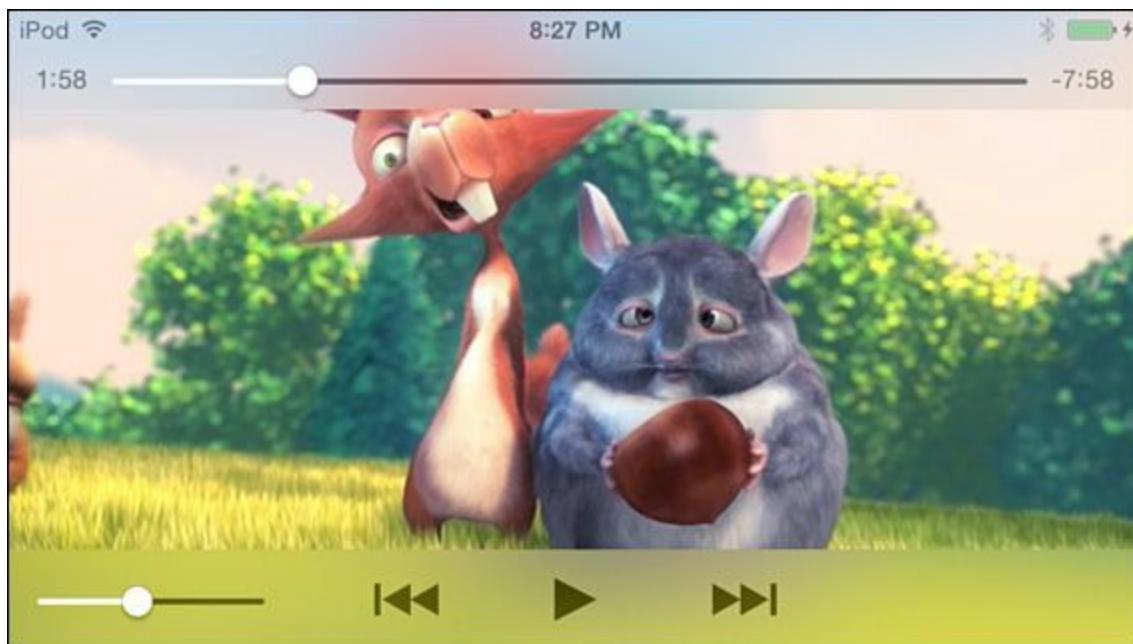


Figure 5.1 AV Kit's AVPlayerViewController

As you can see in [Figure 5.1](#), `AVPlayerViewController` provides a lot of functionality without writing a lot of code. It created a fully functioning player with the same user interface and experience as the default iOS video player. Because this class is a `UIViewController` subclass, it can easily be embedded as a child view controller or presented like any other view controller.

Note

The example shows creating an `AVPlayer` using its `playerWithURL:` convenience initializer. This certainly works for simple cases, but you'll probably be inclined to set up the full playback stack the way you did in the previous chapter when you need more control.

`AVPlayerViewController`, unlike `MPMoviePlayerController`, does not provide a `controlsStyle` property that enables you to specify the playback controls to use. Instead, it provides *dynamic playback controls* that dynamically update to present the needed user interface for the content being played. This means a user is always presented with an appropriate user experience, whether playing a local video with chapter markers or streaming video with subtitles. [Figure 5.2](#) shows some of the alternate controls styles in

action.

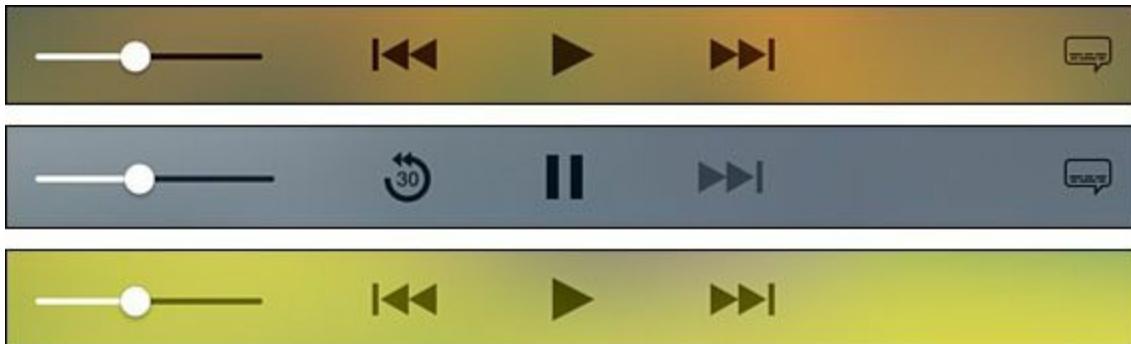


Figure 5.2 `AVPlayerViewController`: dynamic playback controls

`AVPlayerViewController` is a simple but powerful new class available in iOS 8. It provides a great alternative to using `MPMoviePlayerController` because it exposes the full AV Foundation stack, which enables you to use the more advanced features the framework provides. For some additional information on `AVPlayerViewController`, you may be interested in watching *Session 503: Mastering Modern Media Playback* from WWDC 2014.

AV Kit for Mac OS X

AV Kit for Mac was first introduced in Mac OS X Mavericks. The framework provides a class called `AVPlayerView` that makes it really easy to incorporate full-featured video playback into your Mac applications. It provides a video playback experience identical to that of QuickTime Player X, enabling you to easily provide the behavior and user interface Mac users have come to expect.

`AVPlayerView` is an `NSView` subclass used to display and control the playback of an `AVPlayer` instance. Everything you learned in the previous chapter still applies when you're using `AVPlayerView`, but it makes it much faster and easier to develop a video playback application providing standardized playback controls and behaviors. It also provides automatic support for all the standard modern OS X features, such as localization, state restoration, full-screen playback, high-resolution display, and accessibility.

First Steps

In this section you'll learn to use `AVPlayerView` by building an AV Kit-

based video player. In the Chapter 5 directory you'll find a sample project called **KitTime Player_starter**. Although most of the book's sample apps factor out the AV Foundation code into its own set of classes, because this is a Mac-specific topic, you'll develop this app in the context of the main NSDocument instance. Open the project and let's begin.

The first thing to configure is to make some changes to the KitTime Player target's Document Types settings. Highlight the root node of the project in the Project Navigator and select the KitTime Player target. Select the Info tab and expand the Document Types section and make the following changes, as shown in [Figure 5.3](#).

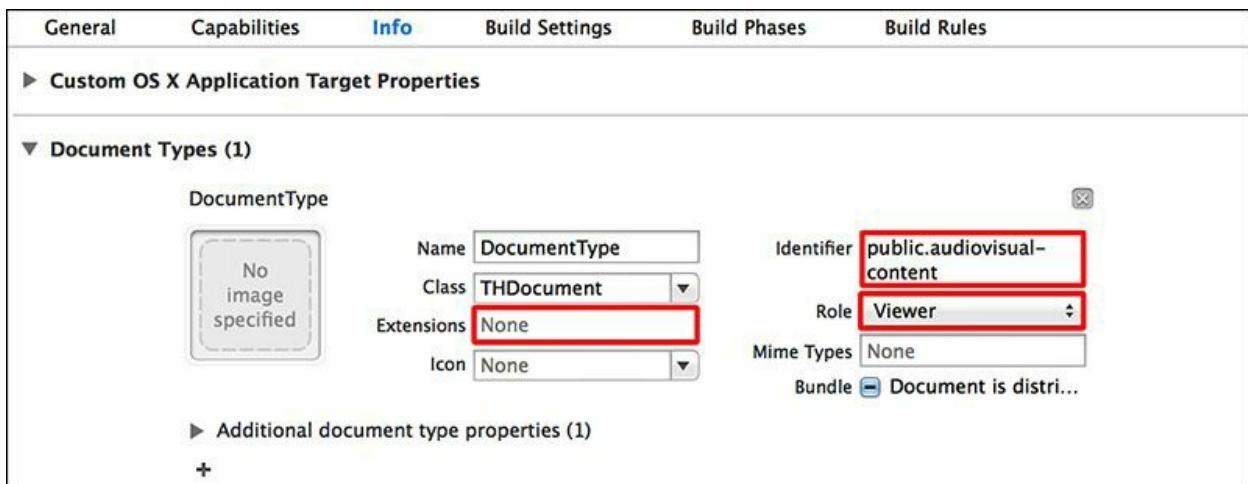


Figure 5.3 KitTime Player's Document Types configuration

- Remove the `mydoc` extension from the Extensions field, because the application won't be creating custom document types.
- Set the Identifier field to `public.audiovisual-content`. Specifying this Uniform Type Identifier will enable the application to open any audiovisual media.
- Set the Role to Viewer so a new document window isn't created when starting the application.

Next, in the Project Navigator select the `THDocument.xib`. The `NSWindow` has been sized to 640×360 to accommodate the aspect ratio of the video content you'll be playing, but feel free to adjust this as you see fit. In the Object Library's search box, begin typing `AVPlayerView` until you see the AV Player View component. Drag an instance of this onto the window and center it within the window's bounds, as shown in [Figure 5.4](#).

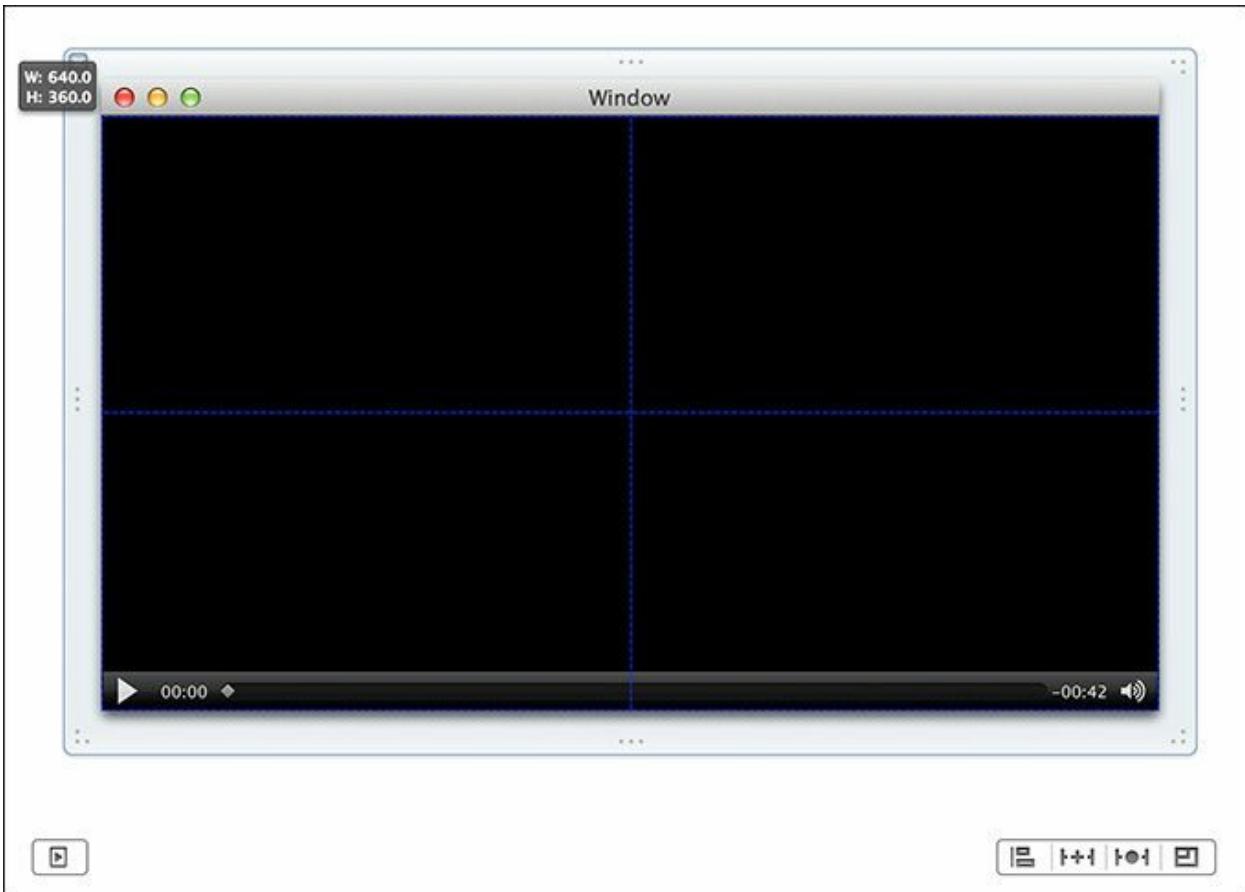


Figure 5.4 Adding an AVPlayerView to the window

To ensure that the AVPlayerView is sized appropriately when you change the window size, you'll need to add the appropriate Auto Layout constraints. The easiest way to do this in this case is to select the Resolve Auto Layout Issues button and select Add Missing Constraints in Window, as shown in [Figure 5.5](#).

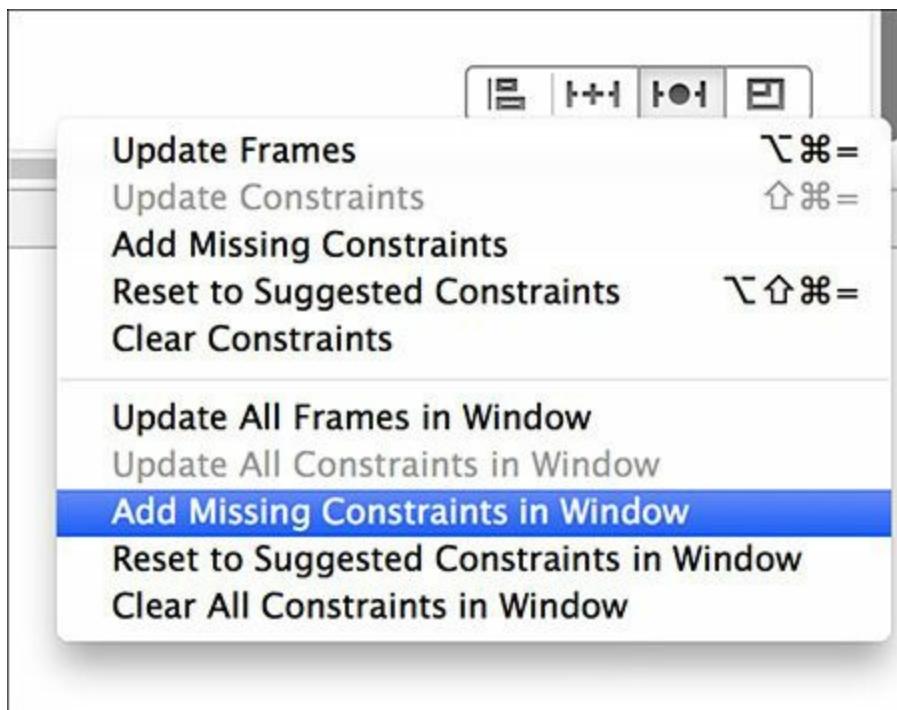


Figure 5.5 Setting the Auto Layout constraints

In the Attributes Inspector for the player view, set its Controls Style property to Floating, as shown in [Figure 5.6](#).

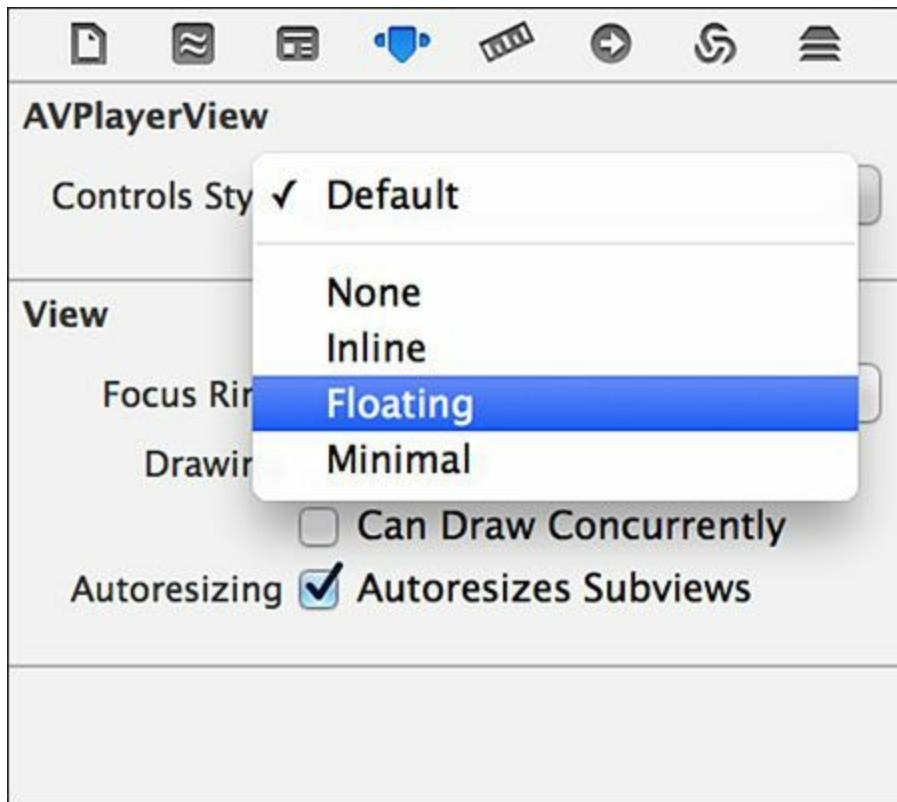


Figure 5.6 Setting the Controls Style property

This provides the same interface that you've seen in QuickTime Player. The THDocument instance already defines an `IBOutlet` for the player view, but you'll need to connect the player to the outlet. Control-drag from File's Owner proxy to the `AVPlayerView` instance and select the `playerView` outlet, as shown in [Figure 5.7](#).

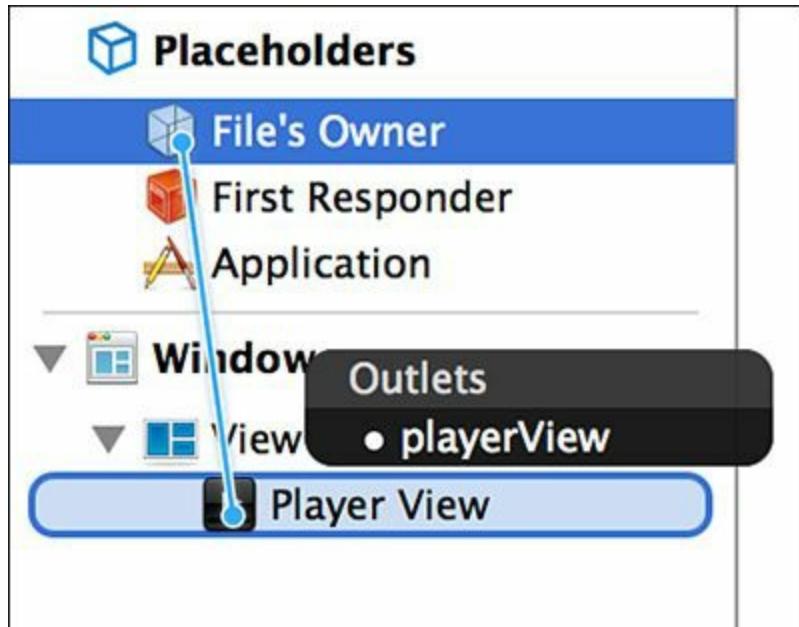


Figure 5.7 Connecting the `playerView` outlet

The required Interface Builder configuration is complete, so switch over to the `THDocument.m` file. [Listing 5.1](#) shows the initial implementation of this class.

Listing 5.1 THDocument Implementation

[Click here to view code image](#)

```
#import "THDocument.h"
#import <AVFoundation/AVFoundation.h>
#import <AVKit/AVKit.h>

@interface THDocument : NSObject
@property (weak) IBOutlet AVPlayerView *playerView;
@end

@implementation THDocument
```

```

- (void)windowControllerDidLoadNib:(NSWindowController
*)aController {
    [super windowControllerDidLoadNib:aController];

}

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

- (BOOL)readFromURL:(NSURL *)url
    ofType:(NSString *)typeName
    error:(NSError * __autoreleasing *)outError {
    return YES;
}

@end

```

This is a bare-bones subclass of `NSDocument`, but it has all the key elements required for building your video player application. The only thing missing is the AV Foundation code. Add the following lines of code to the end of the `windowControllerDidLoadNib:` method.

[Click here to view code image](#)

```

self.playerView.player = [AVPlayer playerWithURL:[self fileURL]];
self.playerView.showsSharingServiceButton = YES;

```

With those lines of code in place, go ahead and run the application. When the application launches, go to the File menu, select Open, and select the `hubblecast.m4v` file in the Chapter 5 directory and press the application's Play button. In only a few minutes of your time and a couple lines of code, you've built the player shown in [Figure 5.8!](#)



Figure 5.8 KitTime Player

`AVPlayerView` automatically provides standard playback controls, a video scrubber, a volume control, dynamic chapter and subtitles menus, and a sharing service menu enabling you to share with the standard destinations provided by Mac OS X. Clearly, AV Kit is a *great* addition to the Mac platform. There's a lot more that you can and will do as we go along, but this is a pretty impressive start. Let's turn our attention for a moment to the various control styles offered by `AVPlayerView`.

Control Styles

`AVPlayerView` provides a number of control styles from which you can choose. You can change this property visually in Interface Builder, as you did previously, or programmatically by modifying its `controlsStyle` property. Let's take a look at the various styles it provides.

Inline

The *Inline* style (`AVPlayerViewControlsStyleInline`) is the default style used by `AVPlayerView` (see [Figure 5.9](#)). This style looks very similar to the interface provided by the QTKit Framework's `QTMovieView`

and provides standard playback, scrubbing, and volume controls. It also supports dynamic chapter and subtitle menus that will be displayed when that data is present in the media.

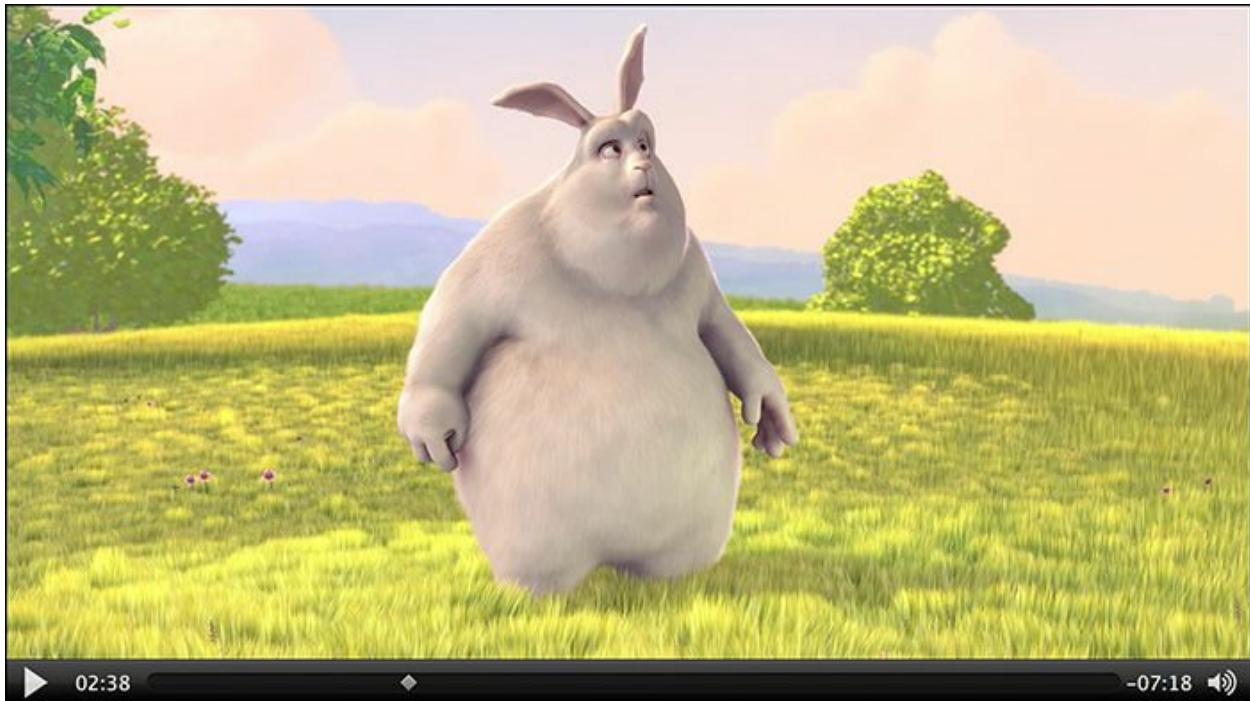


Figure 5.9 AVPlayerView with inline controls

Floating

The *Floating* style (`AVPlayerViewControlsStyleFloating`) provides an appearance identical to that provided by the current version of QuickTime Player (see [Figure 5.10](#)). In fact, the Mavericks version of QuickTime Player is using `AVPlayerView`, so it provides an interface that most Mac users will be immediately familiar using. Like the `Inline` style, it provides standard playback, scrubbing, and volume controls, and if the video contains chapters and subtitles, it will automatically show chapter and subtitle menus as well.

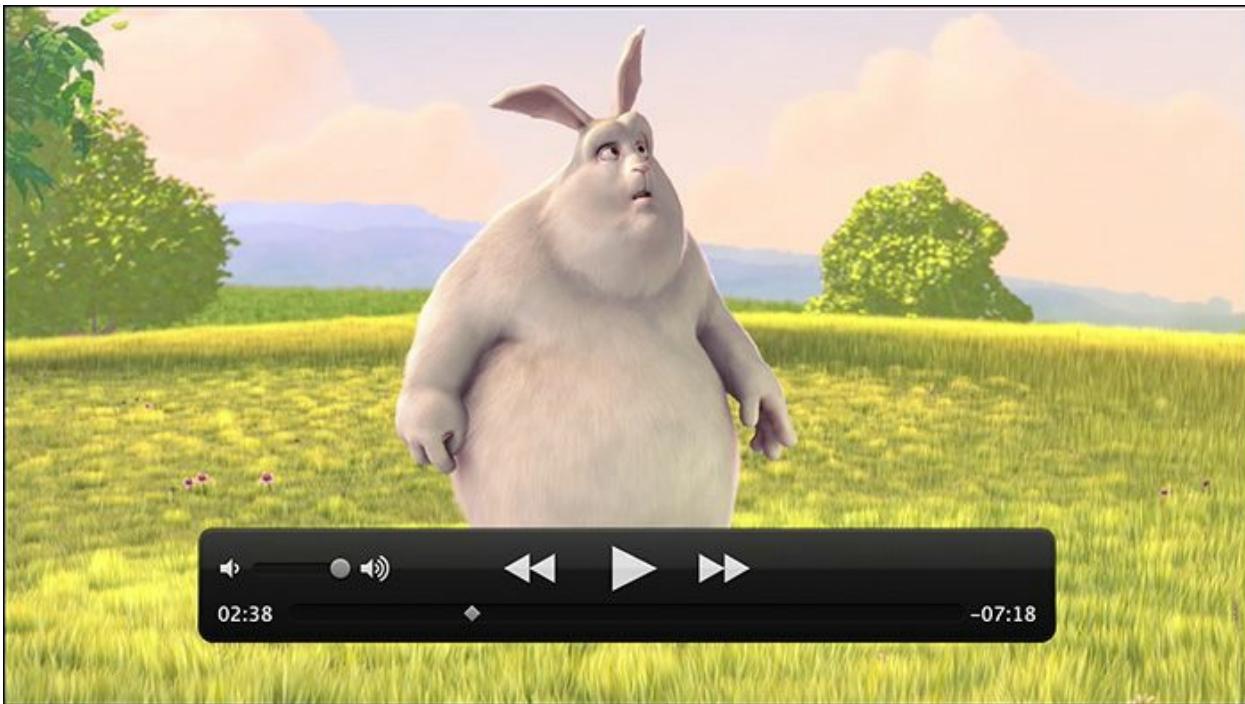


Figure 5.10 AVPlayerView with floating controls

Minimal

The *Minimal* style (`AVPlayerViewControlsStyleMinimal`) provides a floating round button in the center of the screen displaying either a Play or a Pause button along with a circular progress indicator (see [Figure 5.11](#)). This may be an appropriate style to choose when playing short videos requiring minimal control.

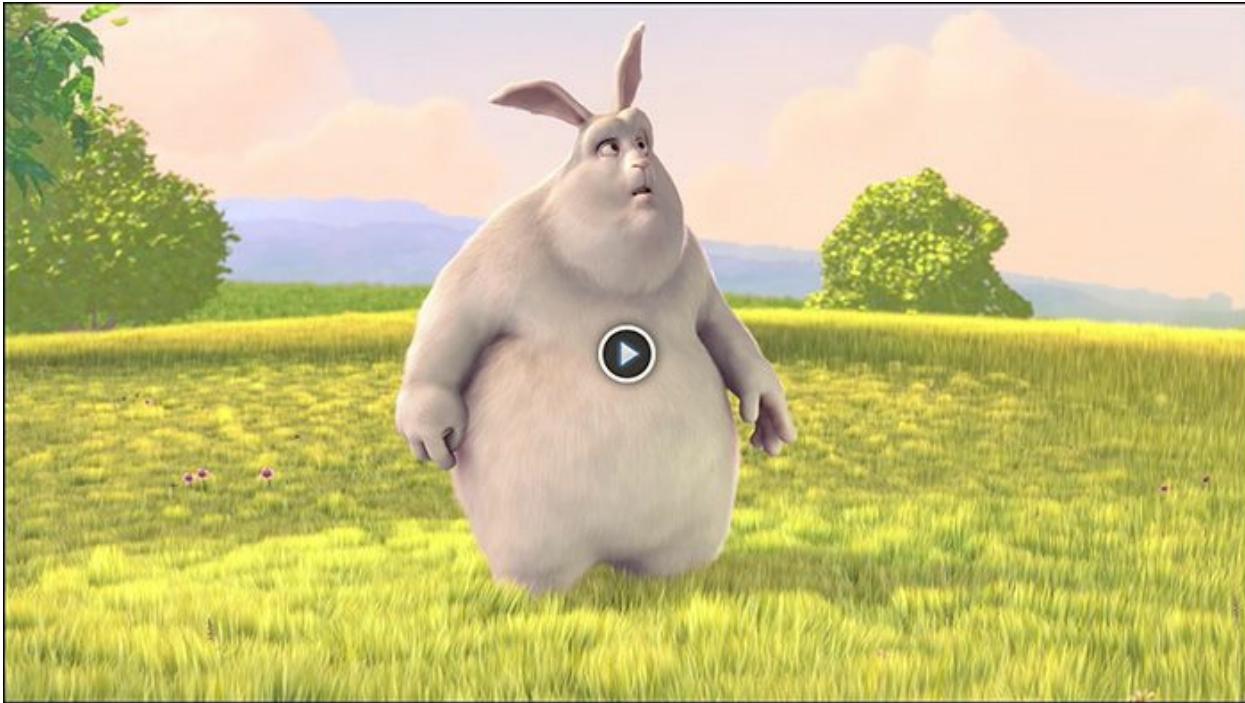


Figure 5.11 AVPlayerView with minimal controls

None

The `None` style (`AVPlayerViewControlsStyleNone`) is essentially no style. Selecting this style provides no controls and simply displays the video content (see [Figure 5.12](#)). This can be useful if you wanted to provide your own playback controls or in scenarios where no explicit control of the video is needed.



Figure 5.12 AVPlayerView with no controls

Regardless of the style selected, `AVPlayerView` always responds to a standard set of keyboard shortcuts. The spacebar plays and pauses video playback. The right and left arrow keys enable you to step frame-by-frame through the video. Additionally, the player view responds to J-K-L navigation where typing the J key rewinds, the L key fast-forwards, and the K key stops playback.

Note

If you are playing an HTTP Live Streaming video, `AVPlayerView` automatically displays alternate controls, according to its controls style, appropriate for streaming playback.

Note

Only the Floating and Inline styles support displaying the Sharing Service menu that you enabled by setting the `showsSharingServiceButton` property to YES.

Going Further

You've seen how you can get up and running quickly with AV Kit, but we want to take things a bit further and enable some more advanced features in this app. To do so, you'll make some modifications starting with the setup of the media stack. `AVPlayer` provides the `playerWithURL:` method, which offers a shortcut way of creating the underlying `AVAsset` and `AVPlayerItem` instances and performing the required preparation of those objects. Using this method can be convenient, but generally when you need to work directly with those objects, you'll likely find it easier to explicitly create and prepare them. You'll use the same basic techniques you did in the previous chapter for setting up the AV Foundation playback stack, as shown in [Listing 5.2](#).

Listing 5.2 Setting up the Playback Stack

[Click here to view code image](#)

```
#import "THDocument.h"
#import <AVFoundation/AVFoundation.h>
#import <AVKit/AVKit.h>
#define STATUS_KEY @"status"

@interface THDocument : NSObject

@property (strong) AVAsset *asset; // 1
@property (strong) AVPlayerItem *playerItem;
@property (strong) NSArray *chapters;

@property (weak) IBOutlet AVPlayerView *playerView;

@end

@implementation THDocument

- (void>windowControllerDidLoadNib:(NSWindowController *)controller {
    [super windowControllerDidLoadNib:controller];

    [self setupPlaybackStackWithURL:[self fileURL]]; // 2
}

- (void)setupPlaybackStackWithURL:(NSURL *)url {
    self.asset = [AVAsset assetWithURL:url];
}
```

```

NSArray *keys = @[@"commonMetadata",
@"availableChapterLocales"];           // 3

    self.playerItem = [AVPlayerItem
playerItemWithAsset:self.asset           // 4
                    automaticallyLoadedAssetKeys:keys];

        [self.playerItem
addObserver:self                           // 5
            forKeyPath:STATUS_KEY
            options:0 context:NULL];

    self.playerView.player = [AVPlayer
playerWithPlayerItem:self.playerItem];
    self.playerView.showsSharingServiceButton = YES;
}

- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {

    if ([keyPath isEqualToString:STATUS_KEY]) {

        if (self.playerItem.status ==
AVPlayerItemStatusReadyToPlay) {
            NSString *title = [self
titleForAsset:self.asset];           // 6
            if (title) {
                self.windowForSheet.title = title;
            }
            self.chapters = [self
chaptersForAsset:self.asset];         // 7
        }

        [self.playerItem removeObserver:self
forKeyPath:STATUS_KEY];               // 8
    }
}

- (NSString *)titleForAsset:(AVAsset *)asset {
    return nil;
}

- (NSArray *)chaptersForAsset:(AVAsset *)asset {
    return nil;
}

```

1. Add three new properties for the `AVAsset`, `AVPlayerItem`, and an `NSArray` to store chapter data that you'll be capturing in an upcoming section.
2. Remove your simple `AVPlayer` creation code and instead factor the set-up code into a new method called `setupPlaybackStackWithURL:` that will be called from the `windowControllerDidLoadNib:` method.
3. Create an `NSArray` to store the `commonMetadata` and `availableChapterLocales` keys. These are the `AVAsset` keys you'll want loaded so you can perform some further inspection of the asset.
4. Create an `AVPlayerItem` using its new `playerItemWithAsset:automaticallyLoadedAssetKeys` convenience initializer. This was a great addition made in Mac OS X 10.9 and iOS 7.0 that simplifies loading `AVAsset` key values.
5. Add `self` as an observer of the player item's `status` property so you can be notified when the player item is ready to play. This happens after the required asset properties have been loaded and the playback pipeline has been primed.
6. If the player item's status is `AVPlayerItemStatusReadyToPlay`, you can safely call the `AVAsset` properties you requested to be loaded.
7. Call `titleForAsset:` so you can retrieve the title for the asset being played. If a valid title is returned, you'll set it as the window's `title` property. If not, the title displays the filename of the asset. You'll provide an implementation for the `titleForAsset:` method in a bit, but for now provide a stub implementation returning `nil`.
8. Call `chaptersForAsset:` to get the array of objects storing data about the chapter markers found in the current asset. We'll discuss how to retrieve chapter data shortly, but for now you'll provide a stub implementation of the `chaptersForAsset:` method.

The changes you made didn't change the functionality but were made in preparation for the upcoming feature enhancements. As a sanity check, run the application again and make sure the functionality continues to work as it

did previously.

Before moving on with the larger features you'll add, let's take a look at how to implement the `titleForAsset:` method (see [Listing 5.3](#)). This method makes use of AV Foundation's standard metadata features you've used in the past two chapters.

Listing 5.3 Implementing `titleForAsset:`

[Click here to view code image](#)

```
- (NSString *)titleInMetadata:(NSArray *)metadata {
    NSArray *items
    =
        [AVMetadataItem metadataItemsFromArray:metadata
                                         forKey:AVMetadataCommonKey
                                         keySpace:AVMetadataKeySpaceCommon];
    return [[items firstObject]
stringValue];                                // 2
}

- (NSString *)titleForAsset:(AVAsset *)asset {
    NSString *title = [self
titleInMetadata:asset.commonMetadata];           // 3
    if (title && (![title isEqualToString:@""])) {
        return title;
    }
    return nil;
}
```

1. Add a new method called `titleInMetadata:` that will extract the title from a collection of `AVMetadataItem` instances. You'll begin by calling the `metadataItemsFromArray:withKey:keySpace:` class method to retrieve the title key in the common key space.
2. Get the `firstObject` from this array and get its value as an `NSString`. The title value in the common key space will always be a string, so you can safely coerce it to an `NSString` using the `stringValue` property.
3. Call the `titleInMetadata:` method passing the asset's `commonMetadata` as the argument. If a valid title was found, you'll

return it to the caller; otherwise, you'll return nil.

You can run the application again. The sample video found in the [Chapter 5](#) directory contains title metadata, so you should see the embedded title in the window's title bar.

Working with Chapters

If you're using either the Floating or Inline control styles, AVPlayerView automatically shows a chapter menu whenever chapter data is present in the video file. This is a great convenience, but what if you need to work directly with this data to implement some additional features, or you're using a controls style that doesn't provide the dynamic chapter menu? Fortunately, AV Foundation enables you to work with this data via the AVTimedMetadataGroup class.

Timed metadata is essentially the same as the static metadata you've been working with since [Chapter 3](#), "[Working with Assets and Metadata](#)," but instead of applying to the asset as a whole, this metadata applies only to certain time ranges within the asset's timeline. AVAsset provides two methods that enable you to retrieve this data:

[Click here to view code image](#)

```
chapterMetadataGroupsWithTitleLocale:containingItemsWithCommonKeys  
chapterMetadataGroupsBestMatchingPreferredLanguages:
```

Both methods return an NSArray of AVTimedMetadataGroup objects representing the chapter metadata found in the asset. As you can probably infer from their method signatures, chapter data is locale dependent. Before you call either method, you first need to make sure the asset's availableChapterLocales key has been loaded, as shown in the following example.

[Click here to view code image](#)

```
NSURL *url = // asset URL;  
AVAsset *asset = [AVAsset assetWithURL:url];  
NSString *key = @"availableChapterLocales";  
  
[asset loadValuesAsynchronouslyForKeys:@[key] completionHandler:^{  
    AVKeyValueStatus status = [asset statusOfValueForKey:key  
error:nil];
```

```

if (status == AVKeyValueStatusLoaded) {

    NSArray *langs = [NSLocale preferredLanguages];
    NSArray *chapterMetadata =
        [asset
    chapterMetadataGroupsBestMatchingPreferredLanguages:langs];

        // Process AVTimeMetadataGroup objects
    }
} ];

```

An AVTimedMetadataGroup contains two properties: timeRange and items. The timeRange property stores a CMTimeRange struct containing a CMTime indicating the time range's start time and a CMTime defining its duration. This enables you to determine the range of time in the asset's timeline to which this metadata applies. The chapter's title, and optionally its thumbnail image, can be found in the items property, which contains an NSArray of AVMetadataItem objects from the common key space.

Let's put this into action by implementing the chaptersForAsset: method in [Listing 5.4](#).

Listing 5.4 Implementing chaptersForAsset:

[Click here to view code image](#)

```

- (NSArray *)chaptersForAsset:(AVAsset *)asset {

    NSArray *languages = [NSLocale
preferredLanguages];                                // 1

    NSArray *metadataGroups
=                                         // 2
        [asset
    chapterMetadataGroupsBestMatchingPreferredLanguages:languages];

    NSMutableArray *chapters = [NSMutableArray array];

    for (NSUInteger i = 0; i < metadataGroups.count; i++) {
        AVTimedMetadataGroup *group = metadataGroups[i];

        CMTime time = group.timeRange.start;
        NSUInteger number = i + 1;
        NSString *title = [self titleInMetadata:group.items];

        THChapter *chapter

```

```

        = // 3
        [THChapter chapterWithTime:time number:number
title:title];

        [chapters addObject:chapter];

    }

    return chapters;
}


```

- 1.** Begin by asking `NSLocale` for its array of `preferredLanguages`. This returns an array of language codes sorted according to the user's languages preferences. In my case, the first element in this list is `en` for English.
- 2.** Ask the asset for its chapter metadata groups that best match the user's preferred languages.
- 3.** Iterate through each of the `AVTimedMetadataGroup` objects and extract the relevant data from each. Specifically, you'll get its start time from its `timeRange` and its title using the `titleInMetadata:` method you defined earlier. You'll also create a chapter number for each chapter based on the current loop index. You'll store this data in a custom model object called `THChapter` that will simplify working with this data.

You've retrieved the chapter metadata and wrapped it up nicely in a collection of `THChapter` objects, so what can you do with this data? You'll put this data into action by utilizing another customization point provided by `AVPlayerView`. An `AVPlayerView` provides an `actionPopUpButtonMenu` enabling you to add a custom `NSMenu` to the player's controls. You'll create a menu providing the capability to skip to the next and previous chapters. Let's start by building the `NSMenu` as shown in [Listing 5.5](#).

Listing 5.5 Creating the Action Menu

[Click here to view code image](#)

```

- (void)setupActionMenu {
    NSMenu *menu = [[NSMenu alloc]
init]; // 1

```

```

[menu addItem:[ [NSMenuItem alloc] initWithTitle:@"Previous
Chapter"
                                         action:@selector(prev:
keyEquivalent:@"""]];
[menu addItem:[ [NSMenuItem alloc] initWithTitle:@"Next
Chapter"
                                         action:@selector(next:
keyEquivalent:@"")];

self.playerView.actionPopUpButtonMenu =
menu; // 2
}

- (void)previousChapter:(id)sender {
}

- (void)nextChapter:(id)sender {
}

```

1. Create a new NSMenu instance and add “Previous Chapter” and “Next Chapter” menu items that call the previousChapter: and nextChapter: selectors, respectively. You’ll provide stub implementations of these methods for the time being.
2. Set this menu as the player view’s actionPopUpButtonMenu property. This menu will be shown when you are using either the Floating or Inline control styles.

Before this menu can be added, you first need to invoke the setupActionMenu method. Modify the observeValueForKeyPath: method, as shown in [Listing 5.6](#).

Listing 5.6 Modifying observeValueForKeyPath:

[Click here to view code image](#)

```

- (void)observeValueForKeyPath:(NSString *)keyPath
                         ofObject:(id)object
                            change:(NSDictionary *)change
                           context:(void *)context {

if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {
    NSString *title = [self titleForAsset:self.asset];
    if (title) {
        self.windowForSheet.title = title;
    }
}

```

```

        }
        self.chapters = [self chaptersForAsset:self.asset];

        // Create action menu if chapters are available
        if ([self.chapters count] > 0) {
            [self setupActionMenu];
        }

    }

    [self.playerItem removeObserver:self forKeyPath:STATUS_KEY];

}

```

You'll want to show this menu only if chapter data exists in the asset. If the `chapters` array isn't empty, you'll call the `setupActionMenu` method.

You can run the application again and verify that the menu is shown. You can click the menu items, but because you haven't yet implemented those methods yet, nothing will happen. Before you can provide implementations for those methods, you'll first need to add some additional support code to find the next and previous chapters (see [Listing 5.7](#)). The following code is reasonably complex given that this is the first time you're seeing some more advanced usage of the Core Media types and functions, so we'll walk through these methods in some detail.

Listing 5.7 Finding Chapters

[Click here to view code image](#)

```

- (THChapter *)findPreviousChapter {

    CMTime playerTime = self.playerItem.currentTime;
    CMTime currentTime = CMTimeSubtract(playerTime, CMTimeMake(3,
1));           // 1
    CMTime pastTime = kCMTimeNegativeInfinity;

    CMTimeRange timeRange = CMTimeRangeMake(pastTime,
currentTime);           // 2

    return [self findChapterInTimeRange:timeRange
reverse:YES];           // 3
}

- (THChapter *)findNextChapter {

    CMTime currentTime =

```

```

self.playerItem.currentTime; // 4
    CMTime futureTime = kCMTimePositiveInfinity;

    CMTimeRange timeRange = CMTimeRangeMake(currentTime,
futureTime); // 5

    return [self findChapterInTimeRange:timeRange
reverse:NO]; // 6
}

- (THChapter *)findChapterInTimeRange:(CMTimeRange)timeRange
reverse:(BOOL)reverse {

    __block THChapter *matchingChapter = nil;

    NSEnumerationOptions options = reverse ? NSEnumerationReverse
: 0;
    [self.chapters
enumerateObjectsWithOptions:options // 7
usingBlock:^(id obj,
NSUInteger idx,
BOOL *stop) {

        if ([(THChapter *)obj isInTimeRange:timeRange])
        // 8
            matchingChapter = obj;
            *stop = YES;
    }
}];

    return
matchingChapter; // 9
}

```

1. To find the previous chapter, start by defining two CMTime values.

The first captures the player item's current time, minus 3 seconds, and the second will be a time infinitely in the past defined using the `kCMTimeNegativeInfinity` constant. When you're playing the video, time is continually marching forward, so you need to give the user a few seconds leeway when selecting the menu item. If you don't, the user would be stuck in a constant loop of going back to the current chapter's start time. When calculating the `currentTime` you'll subtract 3 seconds using the `CMTimeSubtract` function and set the resulting value as the `currentTime`.

2. Create a `CMTimeRange` using the `CMTimeRangeMake` function passing the `pastTime` as the time range's start and the `currentTime` as its duration. This time range will be used to search through the collection of `THChapter` objects.
3. Call the `findChapterInTimeRange:reverse:` method passing YES for the `reverse:` argument. This indicates that you want to search backward through the `chapters` array.
4. Similar to the `findPreviousChapter` method, capture the player item's current time. There's no need to perform any special calculations, because you don't have any timing issues to be concerned with when moving forward. You'll also create a time infinitely in the future using the `kCMTimePositiveInfinity` constant that will mark the upper bound of your time range
5. Create a `CMTimeRange` with the current time as the `start` and the future time as the `duration`.
6. Call `findChapterInTimeRange:reverse:`, this time passing NO as the `reverse:` argument because you want to search forward through the `chapters` array.
7. Enumerate the objects in the `chapters` array, iterating over the collection in the order specified by the `reverse:` argument.
8. Call the chapter's `isInTimeRange:` method to determine if the chapter's start time is in the time range specified. If so, you've found your match and you can stop processing the elements.
9. Finally, you'll return the matching `THChapter`. It is also valid to return `nil` from this method if no match is found, as is the case of when you try to navigate backward at the beginning of the timeline or forward at the end of the timeline.

I won't cover the complete implementation of the `THChapter` class, because it's a simple data holder object, but I do want to call out its `isInTimeRange:` method because it makes use of a helpful macro found in the Core Media framework.

[Click here to view code image](#)

```
- (BOOL)isInTimeRange:(CMTimeRange)timeRange {
    return CMTIME_COMPARE_INLINE(_time, >, timeRange.start) &&
```

```
        CMTIME_COMPARE_INLINE(_time, <, timeRange.duration);  
    }  
}
```

Core Media defines a number of useful functions and macros. One macro you're likely to use fairly regularly is `CMTIME_COMPARE_INLINE`. This macro takes two `CMTime` values along with a comparison operator and returns a Boolean indicating if the comparison is true. In the `isInTimeRange:` method, I'm determining if the chapter's time falls between the time range's start and duration times.

Okay. The hard part is over and you're almost ready to put this feature into action. [Listing 5.8](#) shows the implementation of the remaining methods.

Listing 5.8 Implementing `previousChapter:` and `nextChapter:`

[Click here to view code image](#)

```
- (void)previousChapter:(id)sender {  
    [self skipToChapter:[self  
findPreviousChapter]]; // 1  
}  
  
- (void)nextChapter:(id)sender {  
    [self skipToChapter:[self  
findNextChapter]]; // 2  
}  
  
- (void)skipToChapter:(THChapter *)chapter // 3  
{  
    [self.playerItem seekToTime:chapter.time  
completionHandler:^(BOOL done) {  
        [self.playerView flashChapterNumber:chapter.number  
                                chapterTitle:chapter.title];  
    }];  
}
```

1. When the user selects the Previous Chapter menu, you'll call the `findPreviousChapter` method, passing its result to the `skipToChapter:` method.
2. Similarly, when the user selects the Next Chapter menu, you'll call the `findNextChapter` method and pass the returned value to the `skipToChapter:` method.
3. Finally, in the `skipToChapter:` method, you'll call the player

item's `seekToTime:completionHandler:` method, passing the chapter time as its first argument. In the `completionHandler:callback` you'll call the player view's `flashChapterNumber:chapterTitle:` method which will, you guessed it, flash the chapter number and title on the player view.

Run the application again, open a video, and skip away.

Enabling Trimming

In addition to the great out-of-the-box playback experience provided by `AVPlayerView`, it also has another useful trick up its sleeve. If you've used the current version of QuickTime Player, you may be aware you can go to the Edit menu and select Trim to open a trimming interface, as shown in [Figure 5.13](#).



Figure 5.13 QuickTime Player's trimming interface

`AVPlayerView` provides this same behavior and interface, and the best part is that it's incredibly easy to add to your own application. The sample app already has a Trim menu item in place and a stubbed `startTrimming:` method, but you'll need to provide an implementation for this method, as shown in [Listing 5.9](#).

Listing 5.9 Enabling Trimming

[Click here to view code image](#)

```
- (IBAction)startTrimming:(id)sender {
    [self.playerView beginTrimmingWithCompletionHandler:NULL];
}
```

You may be wondering if I forgot some code, but try it out for yourself. Run the application and open the sample video. Go to the Edit menu and select the Trim menu item. This presents you with the same trimming interface provided by QuickTime Player, and both the Trim and Cancel buttons provide the expected behavior.

Before trimming, you should always begin by querying the player view's canBeginTrimming property. There are a couple cases when this property will return NO. The first is if the trimming interface is already being presented, because it wouldn't make sense to begin trimming in this state. The second is if the asset explicitly disallows trimming. For instance, you are unable to trim movies or TV shows purchased from the iTunes Store. Usually the most appropriate place to query this property would be in the validateUserInterfaceItem: method so you can enable or disable the menu item appropriately (see [Listing 5.10](#)).

Listing 5.10 Providing the Implementation

[Click here to view code image](#)

```
- (BOOL)validateUserInterfaceItem:(id <NSValidatedUserInterfaceItem>)item {
    SEL action = [item action];
    if (action == @selector(startTrimming:)) {
        return self.playerView.canBeginTrimming;
    }
    return YES;
}
```

Now when you run the application, when the trimming interface is presented, the Trim menu item is disabled.

You may be curious how the trimming is being performed. You didn't add any code to trim the asset, and if you recall, `AVAsset` is an immutable object, meaning it can't be modified. So what happened when you clicked the Trim button? Well, an illusion of sorts. When you clicked the Trim button, the player view modifies two properties of the current `AVPlayerItem`. The left side of your trim control sets the player item's `reversePlaybackEndTime` property, and the right side sets the `forwardPlaybackEndTime`. These properties define the asset's effective timeline. If you trim the video and then invoke the Trim menu a second time, you'll see that all the content is still there and the trimming interface is adjusted to the parts you trimmed. This isn't a limitation of `AVPlayerView`, but is simply in keeping with AV Foundation's immutable design philosophy. This is exactly the behavior you'll find in QuickTime Player. So if the asset isn't being modified, how then do you save these changes? This brings us to the topic of exporting.

Exporting

To save the results of your trimming operation, you'll export the current asset as a new one using the `AVAssetExportSession` class. You've seen this class already, and it is one you'll use frequently when working with AV Foundation.

The application already has an Export menu item on the File menu that is hooked up to its corresponding `startExporting:` method. All that's needed is for you to provide an implementation for this method. You'll start by adding a couple new properties to the document object's class extension, as shown in [Listing 5.11](#).

Listing 5.11 Adding Export Properties

[Click here to view code image](#)

```
#import "THEExportWindowController.h"

@interface THDocument () <THEExportWindowControllerDelegate>

@property (strong) AVAsset *asset;
@property (strong) AVPlayerItem *playerItem;
@property (strong) NSArray *chapters;
@property (strong) AVAssetExportSession *exportSession;
@property (strong) THEExportWindowController *exportController;
```

```
@property (weak) IBOutlet AVPlayerView *playerView;  
@end
```

With that set up, let's look at the implementation for the `startExporting:` method, as shown in [Listing 5.12](#).

Listing 5.12 Implementing `startExporting:`

[Click here to view code image](#)

```
- (IBAction)startExporting:(id)sender {  
    [self.playerView.player  
    pause]; // 1  
  
    NSSavePanel *savePanel = [NSSavePanel savePanel];  
  
    [savePanel beginSheetModalForWindow:self.windowForSheet  
    completionHandler:^(NSInteger result) {  
  
        if (result == NSFileHandlingPanelOKButton) {  
            // Order out save panel as the export window will be  
            shown  
            [savePanel orderOut:nil];  
  
            NSString *preset = AVAssetExportPresetAppleM4V720pHD;  
            self.exportSession  
            = // 2  
            [[AVAssetExportSession alloc] initWithAsset:self.asset  
            presetName:preset];  
  
            CMTIME startime =  
            self.playerItem.reversePlaybackEndTime;  
            CMTIME endime =  
            self.playerItem.forwardPlaybackEndTime;  
            CMTIMERange timeRange = CMTIMERangeMake(startime,  
            endime); // 3  
  
            // Configure the export  
            session // 4  
            self.exportSession.timeRange = timeRange;  
            self.exportSession.outputFileType =  
                [self.exportSession.supportedFileTypes  
            firstObject];  
            self.exportSession.outputURL = savePanel.URL;  
  
            self.exportController = [[THEExportWindowController
```

```

alloc] init];
    self.exportController.exportSession =
self.exportSession;
    self.exportController.delegate = self;
    [self.windowForSheet
beginSheet:self.exportController.window // 5
completionHandler:nil];

    [self.exportSession
exportAsynchronouslyWithCompletionHandler:^{
        // Tear
down // 6
    [self.windowForSheet
endSheet:self.exportController.window];
        self.exportController = nil;
        self.exportSession = nil;
    }];
}
}];
}

- (void)exportDidCancel {
    [self.exportSession
cancelExport]; // 7
}

```

1. Start by pausing playback if the video is currently playing, because the user will be unable to interact with the transport controls while the export is running.
2. Create a new instance of `AVAssetExportSession`, passing it the asset and an export preset. In this case I'm using `AVAssetExportPresetAppleM4V720pHD`, which will create a 720p MPEG-4 video file, but feel free to select an alternate preset. For instance, if you're exporting a trimming operation, you may want to consider using `AVAssetExportPresetPassthrough` so you don't have to transcode the media.
3. Create a time range based on the reverse and forward playback end times. If an explicit trim operation hasn't been performed, the reverse and forward end times will be equal to `kCMTimeInvalid`, resulting in a time range equal to `kCMTimeRangeInvalid`. Setting a time range value of `kCMTimeRangeInvalid` on the export session will result in exporting the full duration of the video.

4. Configure the export session with the time range, an output file type compatible with the export session, and the output URL obtained from the NSSavePanel.
5. The sample app provides a class called `THExportWindowController`, which is a simple `NSWindowController` subclass for displaying the export progress window. Create a new instance and set `self` as its delegate and present its window as a new sheet.
6. Start the export, and when its callback is invoked, you'll tear down the export sheet and controller instance.
7. If the user clicks the Cancel button in the export progress window, you'll cancel the export session.

It's time to try out all this functionality. Run the application and open the sample video, trim an interesting section from it, and export a copy to a location on your hard drive. To verify the results of your export you could use... KitTime Player!

Movie Modernization

The one last thing to cover in this chapter is not a part of AV Foundation, but definitely relates to our topic and may be particularly important to you if you have an existing QuickTime-based application.

QuickTime supported a very broad set of codecs and media types, which is part of what made it such a powerful and flexible media platform. However, many of the codecs it supported are no longer relevant and several of its supported track types have been superseded by newer, more capable technologies. AV Foundation does not support these legacy features and instead focuses on what Apple believes are the most relevant codecs and track types for the future.

Given the current media landscape, this focused set of codecs and media types may be entirely reasonable going forward; however, the Macintosh multimedia environment didn't start today. Your users may have years of media encoded with unsupported codecs, and it would be unreasonable to expect them to simply abandon this content. What can you do?

In a rare move, Apple added a new class to a deprecated framework in Mac OS X 10.9. Specifically, it added a new class called `QTMovieModernizer`

to the QTKit framework. The intent of this class is to migrate older media to formats supported by AV Foundation. In fact, you can see this in action if you open an older QuickTime file using the Mavericks version of QuickTime Player. When you open the file you'll see a message saying, "Converting..." prior to opening the media. QuickTime Player is using this class to convert your media, and you can make use of it in your applications as well.

Run the application, select File > Open..., and navigate to the [Chapter 5](#) directory. There you'll find a subdirectory called Legacy containing a few QuickTime files encoded with various legacy codecs. Open one of these files and press the Play button. You can hear the audio coming out of your speakers, but you won't see any video content because AV Foundation has no way of decoding the video track. Let's see how you can make use of the `QTMovieModernizer` class to resolve this issue. You'll begin by making a few modifications to the existing code, as shown in [Listing 5.13](#).

Listing 5.13 Movie Modernization Preparation

[Click here to view code image](#)

```
#import "THDocument.h"
#import <AVFoundation/AVFoundation.h>
#import <AVKit/AVKit.h>
#import "THChapter.h"
#import <QTKit/QTKit.h>
#import "NSFileManager+THAdditions.h"
#import "THWindow.h"

#import "THExportWindowController.h"

#define STATUS_KEY @"status"

@interface THDocument () <THExportWindowControllerDelegate>

@property (strong) AVAsset *asset;
@property (strong) AVPlayerItem *playerItem;
@property (strong) NSArray *chapters;
@property (strong) AVAssetExportSession *exportSession;
@property (strong) THExportWindowController *exportController;

@property BOOL modernizing;

@property (weak) IBOutlet AVPlayerView *playerView;

@end
```

```

@implementation THDocument

#pragma mark - NSDocument Methods

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

- (void)windowControllerDidLoadNib:(NSWindowController *)
    *controller {
    [super windowControllerDidLoadNib:controller];

    if (!self.modernizing) {
        [self setupPlaybackStackWithURL:[self fileURL]];
    } else {
        [(id)controller.window showConvertingView];
    }
}

```

You'll modify the `windowControllerDidLoadNib:` to conditionally perform the setup depending on the state of the `modernizing` property, which indicates whether `QTMovieModernizer` is currently running a conversion. If `modernizing` is false, you'll run the set up as normal, but if it is true, you'll call `showConvertingView` on the `window` object, which displays a progress spinner.

With the changes in [Listing 5.13](#) out of the way, let's move on to the actual modernization process. You'll make the following changes to the `readFromURL:ofType:error:` method, as shown in [Listing 5.14](#).

Listing 5.14 Running the Modernization

[Click here to view code image](#)

```

- (BOOL)readFromURL:(NSURL *)url
    ofType:(NSString *)typeName
    error:(NSError *__autoreleasing *)outError {

    NSError *error = nil;

    if ([QTMovieModernizer requiresModernization:url
error:&error]) {           // 1

        self.modernizing = YES;

        NSURL *destURL = [self
tempURLForURL:url];                      // 2

```

```

        if (!destURL) {
            self.modernizing = NO;
            NSLog(@"Error creating destination URL, skipping
modernization.");
            return NO;
        }

        QTMovieModernizer *modernizer
        =
        [[QTMovieModernizer alloc] initWithSourceURL:url
                                         destinationURL:destURL];

        modernizer.outputFormat =
QTMovieModernizerOutputFormat_H264;           // 6

        [modernizer modernizeWithCompletionHandler:^{
            if (modernizer.status
==                                     // 7
                QTMovieModernizerStatusCompletedWithSuccess) {

                dispatch_async(dispatch_get_main_queue(), ^{
                    [self
setupPlaybackStackWithURL:destURL];           // 8
                    [(id)self.windowForSheet hideConvertingView];
                });
            }
        }];
    }

    return YES;
}

- (NSURL *)tempURLForURL:(NSURL *)url {

    NSFileManager *fileManager = [NSFileManager defaultManager];
    NSString *dirPath
    =
    [fileManager
temporaryDirectoryWithTemplateString:@"kittime.XXXXXX"];

    if (dirPath)
{                                     // 4
        NSString *filePath =
        [dirPath stringByAppendingPathComponent:[url
lastPathComponent]];
        return [NSURL fileURLWithPath:filePath];
    }

    return nil;
}

```

}

1. Begin by determining if the media requires conversion by calling a class method on `QTMovieModernizer` called `requiresModernization:error:`. This method will determine whether a conversion is required. Be aware that this is a synchronous call, so in a real application you would likely want to perform this lookup on a background thread so as not to block the user interface. If this method returns YES, you'll continue on and perform the conversion. In some cases where the return value is YES, the error pointer may contain a description of the error that would occur if you attempted to run the modernization, so you'll want to consider this in your production code.
2. Retrieve an `NSURL` in the `NSTemporaryDirectory()` where you'll write the converted file. If a valid URL could not be retrieved, exit out of the method, which will show the user a default alert message indicating the file could not be read.
3. The `temporaryDirectoryWithTemplateString:` is a category method I added to `NSFileManager`. This method makes use of the `mkdtemp` function to create a unique directory under `NSTemporaryDirectory()` based on the template string passed to the method.
4. If the temporary directory was created, create a new `NSURL` based on the source URL's filename and return it to the caller.
5. Create a new instance of `QTMovieModernizer`, passing it the source and destination URLs.
6. Specify an output format for the converted media. In this case you'll specify h.264 encoding, but you can also specify either Apple ProRes 422 or 4444 using the `QTMovieModernizerOutputFormat_AppleProRes422` or `QTMovieModernizerOutputFormat_AppleProRes4444` constants.
7. Begin the modernization process by calling its `modernizeWithCompletionHandler:` method. When the completion handler block is invoked, you'll determine its status to see

if it was successful. If so, you'll dispatch back to the main thread to continue processing.

8. Call the `setupPlaybackStackWithURL:` method passing the URL of the converted asset. Finally, you'll dismiss the converting view that was previously shown.

Run the application again and reopen one of the sample legacy files. This time you'll see a spinner view while the conversion is underway. When this view is dismissed, you can press the Play button and see the video content thanks to `QTMovieModernizer`.

Summary

In this chapter you gained a good understanding of how to use the AV Kit framework. AV Kit for iOS provides `AVPlayerViewController`, making it easy to build modern video playback apps for iOS using AV Foundation. AV Kit for Mac OS X provides `AVPlayerView`, which provides a simple, yet powerful way of building video playback apps on the Mac and fills the void left by the deprecation of `QTMovieView`. AV Kit enables you to easily create video players with the same features, functionality, and interfaces users have come to expect on both platforms. We also looked at using the `AVTimeMetadataGroup` class to work with timed chapter metadata stored in an `AVAsset`. Although we covered this topic in the context of a Mac application, this functionality can, of course, be used on iOS as well. Finally, we saw the very useful `QTMovieModernizer` class. Although this class is part of the QTKit framework, it can be used to enhance the capabilities of your AV Foundation applications on the Mac platform.

Challenge

KitTime Player provides some of the core features provided by QuickTime Player. Using what you've learned up to this point, take things further by trying to mimic some of QuickTime Player's other features. For instance, experiment with `AVAssetExportSession` to implement some of QuickTime Player's more advanced export functionality. Move the chapter skipping functionality to the View menu and add the appropriate key bindings. Add an Open Location menu item enabling you to play remote media. You may even want to try replicating QuickTime Player's audio recording capabilities using the audio classes covered in [Chapter 2](#), “[Playing](#)

and Recording Audio.”

II: Media Capture

[6 Capturing Media](#)

[7 Using Advanced Capture Features](#)

[8 Reading and Writing Media](#)

6. Capturing Media

The iPhone's introduction in 2007 had a dramatically disruptive effect on the mobile phone industry and completely redefined our understanding of mobile computing. What wasn't immediately obvious at its introduction is how significantly it would also disrupt the digital camera industry. The iPhone has become one of the most popular digital cameras on the market¹ and for many people has completely replaced the traditional point-and-shoot camera and camcorder. This move away from traditional camera devices has also changed *how* we use digital photos and video. No longer are our photos relegated to a photo album and our videos to the living room. Today we can capture, edit, and share our media with the world in only a few taps of our fingers. In this chapter we'll begin our exploration of AV Foundation's capture features to see how you can put them into action to build the next great photo or video application.

1. <http://www.flickr.com/cameras>

Capture Overview

AV Foundation's photo and video capture capabilities have been a strong suit of the framework since its inception. Its introduction in iOS 4 provided developers direct access to an iOS device's cameras and the data they produce, giving rise to a new class of photo and video applications. The framework's capture capabilities continue to be a major area of focus for the media engineers at Apple, with each new release bringing significant new features and enhancements. Although the core capture classes are consistent across iOS and OS X, this is one area of the framework where you will find a few differences. These differences are generally due to features that are tailored to a particular platform. For instance, Mac OS X provides the `AVCaptureScreenInput` class used for screen recording, whereas iOS does not due to its sandboxing restrictions. Our coverage of the AV Foundation's capture features will focus on iOS, but the majority of our discussion will apply to OS X as well.

There are quite a number of classes involved when developing a capture application. The first step to getting started with the framework's features is to gain an understanding of the classes involved and the role and

responsibilities of each. [Figure 6.1](#) provides an overview of the classes used when developing capture apps.

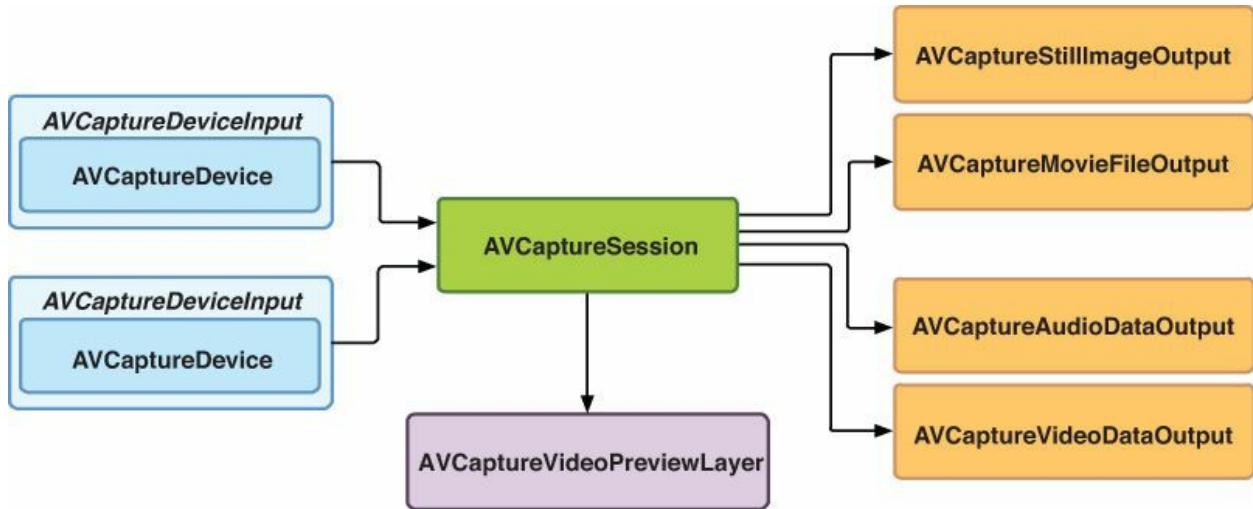


Figure 6.1 AV Foundation capture classes

Capture Session

The central class in AV Foundation’s capture stack is `AVCaptureSession`. A capture session acts as the virtual “patch bay” to which inputs and outputs are connected. It controls the flow of data from physical devices, like cameras and microphones, to one or more output destinations. The routing of inputs and outputs can be configured on-the-fly, enabling you to reconfigure the capture environment as needed over the duration of the session.

The capture session can additionally be configured with a *session preset*, which controls the format and quality of the captured data. The session defaults to a preset value of `AVCaptureSessionPresetHigh`, which works well for many use cases, but there are a variety of presets available to tailor the output to your application’s particular needs.

Capture Devices

`AVCaptureDevice` provides an interface to a physical device such as a camera or a microphone. Most commonly, these devices are integrated into your Mac, iPhone, or iPad, but could also represent an external digital camera or camcorder. An `AVCaptureDevice` provides considerable control over the physical hardware, such as the capability to control the camera’s focus, exposure, white balance, and flash.

`AVCaptureDevice` provides a number of class methods used to get access to the system's capture devices. One method used frequently is `defaultDeviceWithMediaType:`, which returns the system-designated default device for the given media type. Let's look at an example.

[Click here to view code image](#)

```
AVCaptureDevice *videoDevice =  
    [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];
```

In this example we're asking for the default video device. On an iOS device with both front and back cameras, this call returns the back-facing camera because it's designated by the system as its default camera. On camera-equipped Macs, this returns the built-in FaceTime camera.

Capture Device Inputs

Before you can do anything particularly interesting with a capture device, it first needs to be added as an input to the capture session. However, a capture device can't be directly added to an `AVCaptureSession`, but must be wrapped in an instance of `AVCaptureDeviceInput`. This object acts as a virtual patch cable connecting the device's output to the capture session. An `AVCaptureDeviceInput` is created using its `deviceInputWithDevice:error:` method, as shown in the following example.

[Click here to view code image](#)

```
NSError *error;  
  
AVCaptureDeviceInput *videoInput =  
    [AVCaptureDeviceInput deviceInputWithDevice:videoDevice  
error:&error];
```

You should pass a valid `NSError` pointer to this method, because it will be populated with a description of any errors encountered during the input's creation.

Capture Outputs

AV Foundation provides a number of classes extending `AVCaptureOutput`. This is the abstract base class representing an output destination for the data from a capture session. The framework provides some higher-level extensions of this class, such as

`AVCaptureStillImageOutput` and `AVCaptureMovieFileOutput`, which make it easy to capture still images and videos. You'll also find lower-level extensions, such as `AVCaptureAudioDataOutput` and `AVCaptureVideoDataOutput`, which provide direct access to the digital samples captured by the hardware. Using these lower-level outputs requires a greater understanding of the data vended by the capture devices, but they provide you with powerful capabilities, such as performing real-time processing of audio and video streams.

Capture Connections

One class not specifically named in the illustration, but represented by the black arrows connecting the various components, is `AVCaptureConnection`. The capture session determines the media type or types vended by a given capture device input and automatically forms connections to capture outputs that accept media of that type. For instance, an `AVCaptureMovieFileOutput` accepts both audio and video data, so the session will determine which inputs produce video, which produce audio, and wire the connections appropriately. Having access to these connections provides you lower-level control over the signal flow, such as the ability to disable a particular connection or, in the case of an audio connection, access to its individual audio channels.

Capture Preview

A camera application wouldn't be particularly useful if you couldn't see what the camera is capturing. Fortunately, the framework provides the `AVCaptureVideoPreviewLayer` class to fill this need. The preview layer is a Core Animation `CALayer` subclass that provides a real-time preview of the captured video data. This class plays a similar role to that of `AVPlayerLayer`, but is specifically tailored to the needs of camera capture. Like an `AVPlayerLayer`, an `AVCaptureVideoPreviewLayer` supports the notion of video gravities that control how the content it renders is scaled or stretched within the bounds of the layer, as shown in [Figures 6.2, 6.3, and 6.4](#).

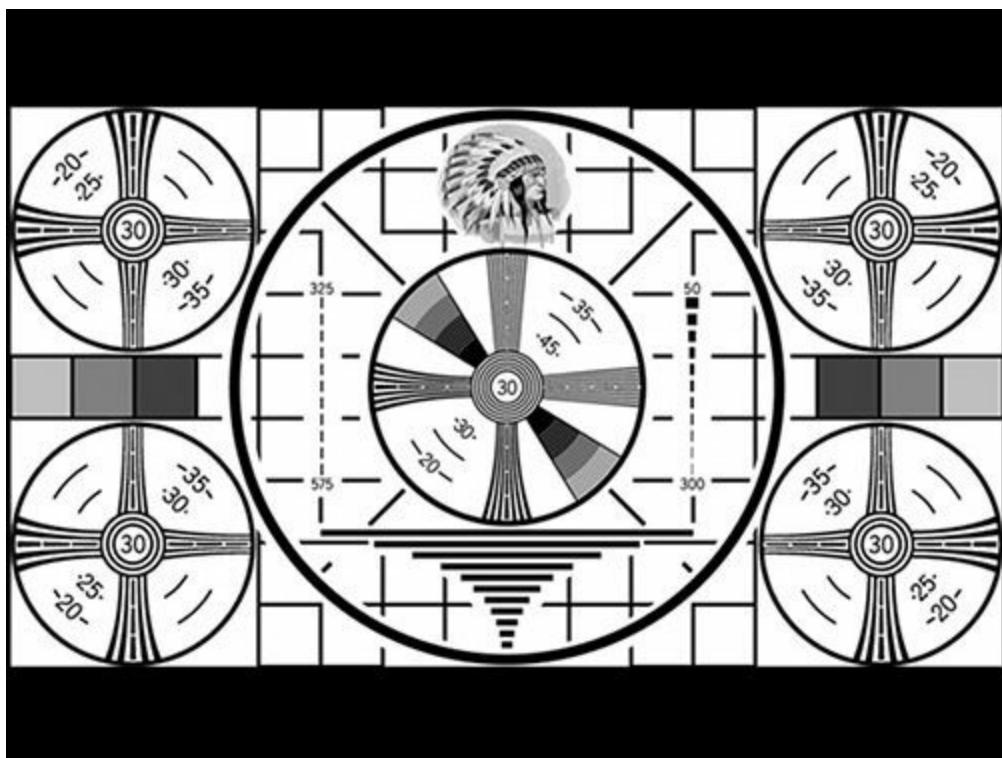


Figure 6.2 `AVLayerVideoGravityResizeAspect` scales the video within the containing layer's bounds to preserve the video's original aspect ratio. This is the default value if not otherwise set and is appropriate in most cases.

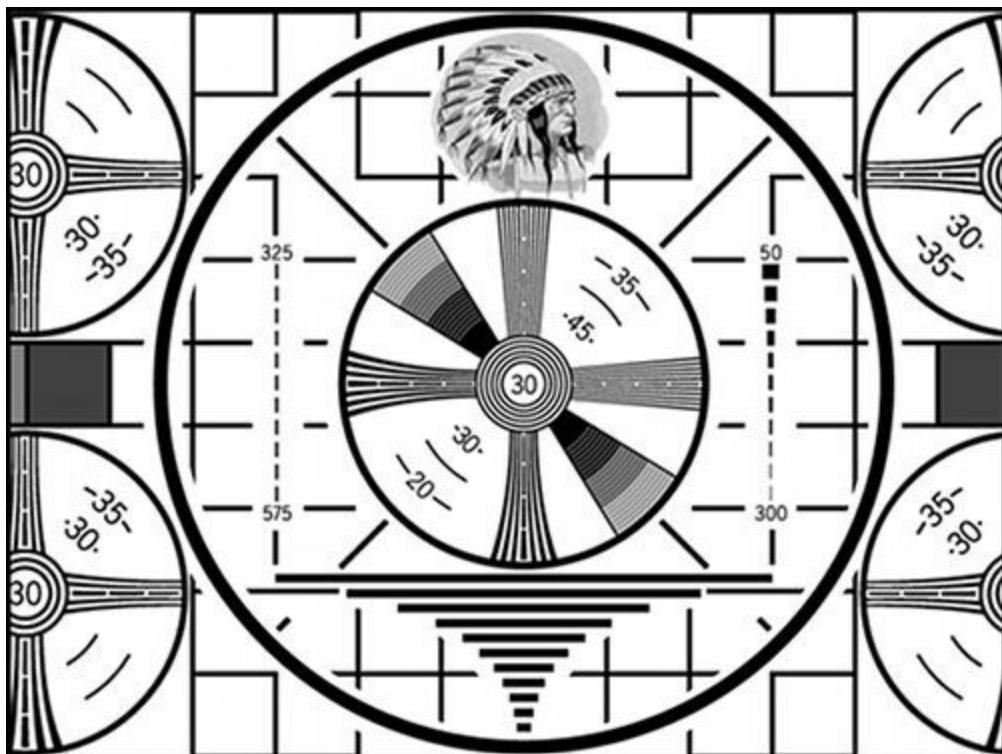


Figure 6.3 `AVLayerVideoGravityResizeAspectFill` preserves the video's aspect ratio while scaling to fill the layer's bounds, often resulting in clipping of the video image.

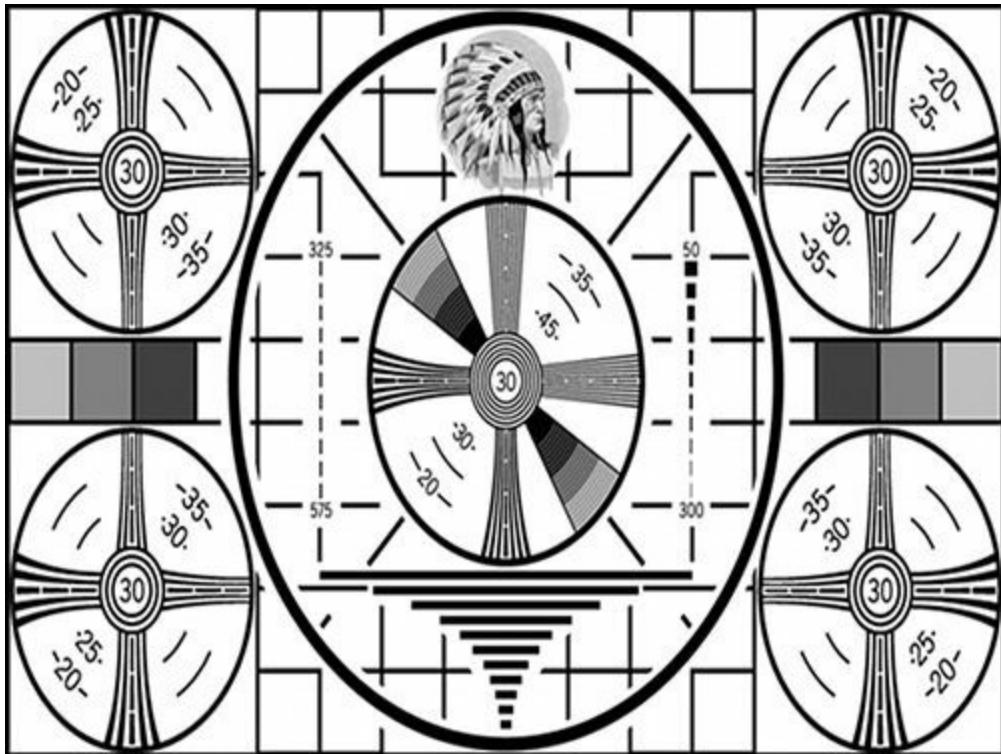


Figure 6.4 `AVLayerVideoGravityResize` stretches the video content to match the containing layer's bounds. This is generally the least useful because it typically results in a “funhouse effect” by distorting the video image.

Simple Recipe

To move beyond the high-level view of the capture classes, let's take a look at an example of how you'd set up a capture session for a simple camera app.

[Click here to view code image](#)

```
// 1. Create a capture session.  
AVCaptureSession *session = [[AVCaptureSession alloc] init];  
  
// 2. Get a reference to the default camera.  
AVCaptureDevice *cameraDevice =  
    [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];  
  
// 3. Create a device input for the camera.  
NSError *error;  
AVCaptureDeviceInput *cameraInput =
```

```

    [AVCaptureDeviceInput deviceInputWithDevice:cameraDevice
error:&error];

// 4. Connect the input to the session.
if ([session canAddInput:cameraInput]) {
    [session addInput:cameraInput];
}

// 5. Create an AVCaptureOutput to capture still images.
AVCaptureStillImageOutput *imageOutput =
    [[AVCaptureStillImageOutput alloc] init];

// 6. Add the output to the session.
if ([session canAddOutput:imageOutput]) {
    [session addOutput:imageOutput];
}

// 7. Start the session and begin the flow of data.
[session startRunning];

```

The example sets up the basic infrastructure required for capturing still images from the default camera. It creates a capture session, adds the default camera to the session via a capture device input, adds a capture output to the session capable of outputting still images, and then starts running the session, which causes the video data to begin flowing through the system. Your typical session setup will often be a bit more involved than this example, but it provides a good illustration of how the core components fit together.

Now that you have a general understanding of the core capture classes, let's begin digging into the details of AV Foundation's capture capabilities by putting these classes into action.

Building a Camera App

In this section we'll jump into the details of AV Foundation's capture APIs by building a camera app called Kamera (see [Figure 6.5](#)). The sample project, patterned after Apple's built-in Camera app, will enable you to capture high-quality still images and videos and write them to the iOS Camera Roll using the Assets Library framework. Along the way you'll implement a number of useful enhancements to this core behavior that will provide you with a good understanding of the capabilities of AV Foundation's capture classes. You'll find a starter project in the Chapter 6 directory called **Kamera_Starter**.



Figure 6.5 Kamera app user interface

Note

Developing the Kamera app will require you to build and test on a real device. Many of AV Foundation's capabilities can be tested using the iOS Simulator, but the capture APIs can be tested only on an iOS device.

Creating the Preview View

[Figure 6.6](#) provides an illustration of how the Kamera app's user interface is composed. We'll focus our attention on the implementation of the middle layer, THPreviewView, because it's the one component of the user

interface directly involving AV Foundation.



Figure 6.6 User interface layers

The `THPreviewView` class shown in [Figure 6.6](#) provides the user a real-time preview of what the camera is currently capturing. You'll make use of `AVCaptureVideoPreviewLayer` to implement this behavior. Let's look at how `THPreviewView` is implemented by starting with its interface, as shown in [Listing 6.1](#).

Listing 6.1 `THPreviewView` Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>

@protocol THPreviewViewDelegate <NSObject>
- (void)tappedToFocusAtPoint:(CGPoint)point;
- (void)tappedToExposeAtPoint:(CGPoint)point;
```

```

- (void)tappedToResetFocusAndExposure;
@end

@interface THPreviewView : UIView

@property (strong, nonatomic) AVCaptureSession *session;
@property (weak, nonatomic) id<THPreviewViewDelegate> delegate;

@property (nonatomic) BOOL tapToFocusEnabled;
@property (nonatomic) BOOL tapToExposeEnabled;

@end

```

Most of the properties and methods defined have to do with handling various tap gestures. The Kamera app supports both tap-to-focus and tap-to-expose functionality. However, the key property this class defines is `session`, which will be used to associate the `AVCaptureVideoPreviewLayer` with the active `AVCaptureSession`. Let's move on to the implementation for this class.

[Listing 6.2](#) provides an abbreviated listing for this class. Most of the code you'll find in the project version of this class has to do with touch handling and is omitted from this discussion. Instead, let's look at the code specifically relating to AV Foundation.

Listing 6.2 **THPreviewView** Implementation

[Click here to view code image](#)

```

#import "THPreviewView.h"

@implementation THPreviewView

+ (Class)layerClass
{
    return [AVCaptureVideoPreviewLayer class];
} // 1

- (void)setSession:(AVCaptureSession *)session
{
    // 2
    [(AVCaptureVideoPreviewLayer*)self.layer setSession:session];
}

- (AVCaptureSession*)session {
    return [(AVCaptureVideoPreviewLayer*)self.layer session];
}

```

```

- (CGPoint)captureDevicePointForPoint:(CGPoint)point
{
    // 3
    AVCaptureVideoPreviewLayer *layer =
        (AVCaptureVideoPreviewLayer *)self.layer;
    return [layer captureDevicePointOfInterestForPoint:point];
}

@end

```

1. A `UIView` is always backed by an instance of `CALayer`. Normally this will be a generic instance of `CALayer`, but overriding the `layerClass` class method enables you to customize the layer type used whenever an instance of the view is created. You'll override the `layerClass` method on `UIView` and return the `AVCaptureVideoPreviewLayer` class object.
2. Override the accessors for the `session` property. In the `setSession:` method you access the view's `layer` property, which is an `AVCaptureVideoPreviewLayer` instance, and set the `AVCaptureSession` on it. This enables the capture output to be directed to the layer and ensures it stays in sync with the state of the session. You'll additionally override the `session` method to return the capture session if requested.
3. The `captureDevicePointForPoint:` method is a private method used to support the various touch handling methods this class provides. It will convert a touch point from screen coordinates to camera coordinates.

Coordinate Space Translations

I wanted to call out the `captureDevicePointForPoint:` method in [Listing 6.2](#) to draw your attention to a couple important methods provided by `AVCaptureVideoPreviewLayer`. When working with AV Foundation's capture APIs, it's important to understand the difference between *screen* coordinates and *capture device* coordinates.

An iPhone 5 or 5s has screen coordinates of (0,0) in the upper-left corner and (320,568) in the lower-right corner in portrait mode or (568,320) in landscape mode. As an iOS developer, you're very familiar with working with points in this space. Capture device coordinates are defined differently. They are

always based on the camera sensor's native, unrotated orientation of *landscape* and range from (0,0) in the upper-left to (1,1) in the bottom-right (see [Figure 6.7](#)).



Figure 6.7 Screen coordinates (left); device coordinates (right)

Prior to iOS 6, converting between these two coordinate spaces was quite challenging. To accurately convert a screen point to a camera point (and vice versa), you had to take into consideration things such as video gravity, mirroring, layer transformations, and orientation to properly calculate this translation. Thankfully, `AVCaptureVideoPreviewLayer` now provides conversion methods that make this process trivial.

`AVCaptureVideoPreviewLayer` provides two methods to translate between the two coordinate spaces:

- `captureDevicePointOfInterestForPoint` : takes a `CGPoint` from the screen coordinate space and returns a translated `CGPoint` in the device coordinate space.
- `pointForCaptureDevicePointOfInterest` : takes a `CGPoint` from the camera device coordinate space and returns a translated `CGPoint` in the screen coordinate space.

`THPreviewView` makes use of the `captureDevicePointOfInterestForPoint:` method to translate the user's touch point to camera device space. This translated point will be used when implementing the Kamera app's tap-to-focus and tap-to-expose functionality.

With the implementation of `THPreviewView` complete, let's move on and discuss the core capture code.

Creating the Capture Controller

The capture session code will be contained within a class called `THCameraController`. This class will configure and manage the various capture devices, as well as control and interact with the capture outputs. Let's begin by looking at the interface for the `THCameraController` class in [Listing 6.3](#).

Listing 6.3 `THCameraController` Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>

extern NSString *const THThumbnailCreatedNotification;

@protocol THCameraControllerDelegate
<NSObject>                                // 1
- (void)deviceConfigurationFailedWithError:(NSError *)error;
- (void)mediaCaptureFailedWithError:(NSError *)error;
- (void)assetLibraryWriteFailedWithError:(NSError *)error;
@end

@interface THCameraController : NSObject

@property (weak, nonatomic) id<THCameraControllerDelegate>
delegate;
@property (nonatomic, strong, readonly) AVCaptureSession
*captureSession;

// Session
Configuration                                         //,
2
- (BOOL)setupSession:(NSError **)error;
- (void)startSession;
- (void)stopSession;
```

```

// Camera Device
Support // 3
- (BOOL)switchCameras;
- (BOOL)canSwitchCameras;
@property (nonatomic, readonly) NSUInteger cameraCount;
@property (nonatomic, readonly) BOOL cameraHasTorch;
@property (nonatomic, readonly) BOOL cameraHasFlash;
@property (nonatomic, readonly) BOOL cameraSupportsTapToFocus;
@property (nonatomic, readonly) BOOL cameraSupportsTapExpose;
@property (nonatomic) AVCaptureTorchMode torchMode;
@property (nonatomic) AVCaptureFlashMode flashMode;

// Tap to *
Methods // 4
- (void)focusAtPoint:(CGPoint)point;
- (void)exposeAtPoint:(CGPoint)point;
- (void)resetFocusAndExposureModes;

/** Media Capture Methods // 5
 */

// Still Image Capture
- (void)captureStillImage;

// Video Recording
- (void)startRecording;
- (void)stopRecording;
- (BOOL)isRecording;

@end

```

This is a larger interface than some of the classes you've built so far, so let's break this down a bit to get an understanding of what you'll be implementing.

- 1.** `THCameraControllerDelegate` defines the methods that will be called on this object's delegate in the event of an error.
- 2.** These methods are used to configure and control the capture session.
- 3.** These methods provide the capability to switch between cameras as well as test the camera's various capabilities so the client can make the appropriate user interface decisions.
- 4.** These methods are used by the tap-to-focus and tap-to-expose features, enabling you to set the focus and exposure at various touch points.
- 5.** These methods provide the capability to capture still images and videos.

Let's switch over to the implementation file and begin the implementation.

Setting Up the Capture Session

We'll begin implementing the THCameraController class by starting with its `setupSession:` method, as shown in [Listing 6.4](#).

Listing 6.4 `setupSession:` Method

[Click here to view code image](#)

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>
#import <AssetsLibrary/AssetsLibrary.h>
#import "NSFileManager+THAdditions.h"

@interface THCameraController : NSObject

@property (strong, nonatomic) dispatch_queue_t videoQueue;
@property (strong, nonatomic) AVCaptureSession *captureSession;
@property (weak, nonatomic) AVCaptureDeviceInput
*activeVideoInput;

@property (strong, nonatomic) AVCaptureStillImageOutput
*imageOutput;
@property (strong, nonatomic) AVCaptureMovieFileOutput
*movieOutput;
@property (strong, nonatomic) NSURL *outputURL;

@end

@implementation THCameraController

- (BOOL)setupSession:(NSError **)error {

    self.captureSession = [[AVCaptureSession alloc]
    init]; // 1
    self.captureSession.sessionPreset =
    AVCaptureSessionPresetHigh;

    // Set up default camera device
    AVCaptureDevice *videoDevice
    = // 2
        [AVCaptureDevice
    defaultDeviceWithMediaType:AVMediaTypeVideo];

    AVCaptureDeviceInput *videoInput
    = // 3
        [AVCaptureDeviceInput deviceInputWithDevice:videoDevice
```

```

error:error];
    if (videoInput) {
        if ([self.captureSession canAddInput:videoInput])
{
                // 4
                [self.captureSession addInput:videoInput];
                self.activeVideoInput = videoInput;
}
    } else {
        return NO;
}

// Setup default microphone
AVCaptureDevice *audioDevice
=
[AVCaptureDevice
defaultDeviceWithMediaType:AVMediaTypeAudio]; // 5

AVCaptureDeviceInput *audioInput
=
[AVCaptureDeviceInput deviceInputWithDevice:audioDevice // 6
error:error];
    if (audioInput) {
        if ([self.captureSession canAddInput:audioInput])
{
                // 7
                [self.captureSession addInput:audioInput];
}
    } else {
        return NO;
}

// Set up the still image output
self.imageOutput = [[AVCaptureStillImageOutput alloc]
init]; // 8
self.imageOutput.outputSettings = @{@"AVVideoCodecKey" : AVVideoCodecJPEG};

if ([self.captureSession canAddOutput:self.imageOutput]) {
    [self.captureSession addOutput:self.imageOutput];
}

// Set up movie file output
self.movieOutput = [[AVCaptureMovieFileOutput alloc]
init]; // 9

if ([self.captureSession canAddOutput:self.movieOutput]) {
    [self.captureSession addOutput:self.movieOutput];
}

self.videoQueue =
dispatch_queue_create("com.tapharmonic.VideoQueue", NULL);

```

```
    return YES;  
}
```

1. The first object to create is the session itself. The `AVCaptureSession` is the central hub of activity in a capture scenario and is the object to which inputs and outputs are added. A capture session can be configured with a session preset. In this case you'll set the preset to `AVCaptureSessionPresetHigh`, which is the default and is suitable for the needs of the Kamera app. Additional session presets can be found in the documentation for `AVCaptureSession`.
2. Get a pointer to the *default* video capture device. In almost all cases on iOS, the `AVCaptureDevice` returned will be the back-facing camera.
3. Prior to adding a capture device to an `AVCaptureSession`, it first needs to be wrapped in an `AVCaptureDeviceInput` object. It's particularly important that you pass a valid `NSError` pointer to this method to capture any errors that occur when attempting to create an input.
4. If a valid `AVCaptureDeviceInput` was returned, you'll first want to test that it can be added to the session by calling the session's `canAddInput:` method and, if so, add it to the session by calling `addInput:` and passing it the capture device input.
5. Similar to the device lookup you performed to get the default video capture device, you'll do the same for the default audio capture device. This will return a pointer to the built-in microphone.
6. Create a capture device input for this device. Again, it's very important to be mindful of any errors encountered when calling this method.
7. Test to see if this device can be added to the session and, if so, you'll add it to the capture session.
8. Create an instance of `AVCaptureStillImageOutput`. This is the `AVCaptureOutput` subclass used to capture still images from the camera. You'll configure its `outputSettings` by providing it a dictionary indicating you would like images captured in JPEG format.

After it's created and configured, you'll add it to the capture session by calling the `addOutput :` method. Just as you did when adding your device inputs, you'll also want to test that the specified output can be added to the capture session before doing so. Blindly adding an output without first testing that it can be added may result in an exception being thrown, causing an application crash.

9. Create a new instance of `AVCaptureMovieFileOutput`. This is an `AVCaptureOutput` subclass used to record QuickTime movies to the file system. You'll test and add this output to the session. Finally, you'll return `YES`, letting the caller know the session configuration is complete.

Starting and Stopping the Session

The capture session's object graph will be properly set up upon calling the `setupSession :` method. However, before the capture session can be used, it first needs to be started. Starting the session begins the flow of data and makes it ready to begin capturing images and videos. Let's look at the implementation of the `startSession` and `stopSession` methods in [Listing 6.5](#).

Listing 6.5 Starting and Stopping the Capture Session

[Click here to view code image](#)

```
- (void)startSession {
    if (![self.captureSession isRunning])
    {
        // 1
        dispatch_async(self.videoQueue, ^{
            [self.captureSession startRunning];
        });
    }
}

- (void)stopSession {
    if ([self.captureSession isRunning])
    {
        // 2
        dispatch_async(self.videoQueue, ^{
            [self.captureSession stopRunning];
        });
    }
}
```

-
1. Test to make sure the capture session isn't already running. If it's not, you'll call the capture session's `startRunning` method. This is a synchronous call and can take some time to complete, so you'll asynchronously enqueue this call on the `videoQueue` so that you don't block the main thread.
 2. The implementation for `stopSession` is nearly identical. Calling `stopRunning` on the capture session stops the flow of data through the system. This, too, is a synchronous call, so you'll asynchronously invoke this method.

Run the application. Immediately upon launch, you'll be presented with one or both dialogs shown in [Figure 6.8](#). Let's discuss why these are shown and how to handle these prompts.

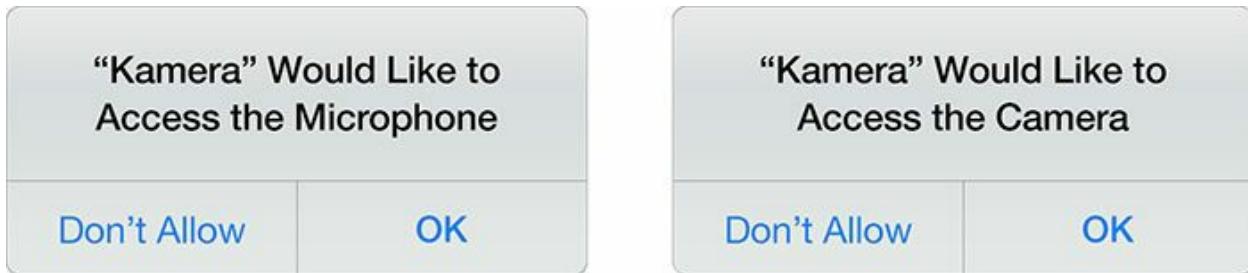


Figure 6.8 Capture device privacy alerts

Handling Privacy Requirements

Additional privacy enhancements have been made in iOS 7 and iOS 8, providing greater transparency about the hardware an application is attempting to use. Users will be presented with dialogs requesting approval whenever an application attempts to use the microphone or the camera. These privacy enhancements are a very welcome addition to the platform, but they do introduce some additional concerns you'll need to address when building capture apps.

Note

iOS 7 requires user approval to access the camera only in regions where it is required by law, such as China. Starting with iOS 8, users will be prompted to grant access to the camera in all regions.

The trigger for these prompts is the creation of an

`AVCaptureDeviceInput`. When these dialogs are presented, the system does not block and wait until the user responds, but instead immediately returns a device that will record silence, in the case of an audio device, or black frames, in the case of a camera device. It's not until the user has answered the prompt in the affirmative that audio or video content will actually be captured. If the user answers Don't Allow to either of these prompts, nothing will be recorded for the duration of the session. If the user disallowed access, the creation of the `AVCaptureDeviceInput` on subsequent launches of the application will return `nil`, and the `NSError` passed to its creation method will be populated with an error code of `ALErrorApplicationIsNotAuthorizedToUseDevice` and a failure reason such as the following:

[Click here to view code image](#)

```
NSLocalizedFailureReason=This app is not authorized to use iPhone Microphone.
```

If you receive this error, the only course of action is to present an error message, such as the one shown in [Figure 6.9](#), indicating the user would need to go to the Settings app and permit access to the required hardware.

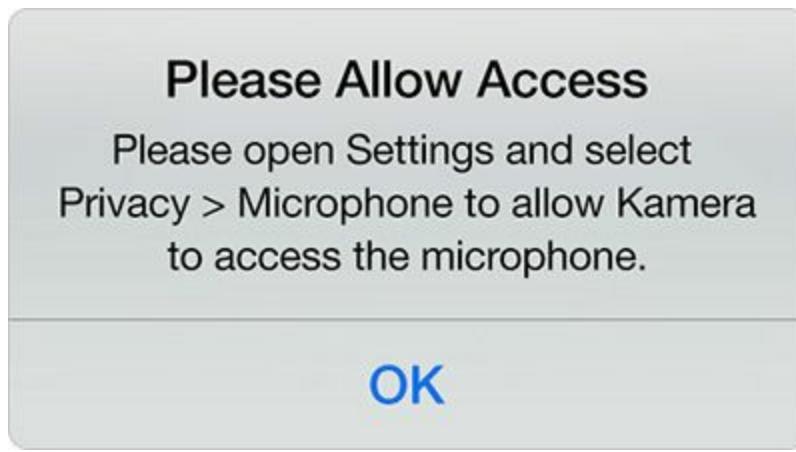


Figure 6.9 Required hardware alert message

Note

Users can change their privacy settings at any time in the Settings app, so it's important that you *always* observe any errors returned when creating an `AVCaptureDeviceInput`.

With the session configuration complete, let's move on and start building out the behavior of the Kamera app.

Switching Cameras

Almost all current generation iOS devices contain both front and back cameras. We'd like to be able to use either in the Kamera app, so the first feature to develop is to provide the user the capability to switch between these cameras. Let's start with some supporting methods that will simplify the implementation of this feature (see [Listing 6.6](#)).

Listing 6.6 Camera Support Methods

[Click here to view code image](#)

```
- (AVCaptureDevice *)cameraWithPosition:  
    (AVCaptureDevicePosition)position { // 1  
    NSArray *devices = [AVCaptureDevice  
        devicesWithMediaType:AVMediaTypeVideo];  
    for (AVCaptureDevice *device in devices) {  
        if (device.position == position) {  
            return device;  
        }  
    }  
    return nil;  
}  
  
- (AVCaptureDevice *)activeCamera  
{  
    return self.activeVideoInput.device; // 2  
}  
  
- (AVCaptureDevice *)inactiveCamera  
{  
    AVCaptureDevice *device = nil;  
    if (self.cameraCount > 1) {  
        if ([self activeCamera].position ==  
            AVCaptureDevicePositionBack) {  
            device = [self  
                cameraWithPosition:AVCaptureDevicePositionFront];  
        } else {  
            device = [self  
                cameraWithPosition:AVCaptureDevicePositionBack];  
        }  
    }  
    return device;  
}
```

```

- (BOOL) canSwitchCameras
{
    return self.cameraCount > 1;                                // 4
}

- (NSUInteger) cameraCount
{
    return [[AVCaptureDevice
devicesWithMediaType:AVMediaTypeVideo] count];                // 5

}

```

1. The `cameraWithPosition:` method returns the `AVCaptureDevice` at the specified position. Valid positions are `AVCaptureDevicePositionFront` or `AVCaptureDevicePositionBack`. You'll iterate over the available video devices and return the one whose position matches the argument value.
2. The `activeCamera` method identifies the camera currently attached to the capture session. You'll return the active capture device input's `device` property.
3. The `inactiveCamera` method returns the inactive camera by looking up the camera at the position opposite the active camera. If the device on which the app is running contains only a single camera, you'll return `nil`.
4. The `canSwitchCameras` method returns a `BOOL` indicating if more than one camera is available.
5. Finally, the `cameraCount` returns a count of all available video capture devices.

With these methods in place, let's continue on with our camera-switching feature. Switching between the front and back cameras involves reconfiguring the capture session. Fortunately, `AVCaptureSession` can be reconfigured on-the-fly, so you don't have to incur the cost of stopping and restarting the session. However, any changes you make to the session should be made as a single, atomic change using its `beginConfiguration` and `commitConfiguration` method pair. Let's see how this is used in [Listing 6.7](#).

Listing 6.7 Switching Cameras

[Click here to view code image](#)

```
- (BOOL)switchCameras {

    if (![self canSwitchCameras]) // 1
    {
        return NO;
    }

    NSError *error;
    AVCaptureDevice *videoDevice = [self
inactiveCamera]; // 2

    AVCaptureDeviceInput *videoInput =
        [AVCaptureDeviceInput deviceInputWithDevice:videoDevice
error:&error];

    if (videoInput) {
        [self.captureSession
beginConfiguration]; // 3

        [self.captureSession
removeInput:self.activeVideoInput]; // 4

        if ([self.captureSession canAddInput:videoInput])
        { // 5
            [self.captureSession addInput:videoInput];
            self.activeVideoInput = videoInput;
        } else {
            [self.captureSession addInput:self.activeVideoInput];
        }

        [self.captureSession
commitConfiguration]; // 6

    } else {
        [self.delegate
deviceConfigurationFailedWithError:error]; // 7
        return NO;
    }

    return YES;
}
```

1. Begin by verifying that you *can* switch cameras. If not, you'll simply return NO and exit the method.
2. Next, get a pointer to the inactive camera and create a new

`AVCaptureDeviceInput` for it.

3. Call `beginConfiguration` on the session to mark the start of an atomic configuration change.
4. Remove the currently active `AVCaptureDeviceInput`. This current video capture device input must be removed before the new one can be added.
5. Perform the standard test to verify that you can add the new `AVCaptureDeviceInput` and, if so, add it to the session and set it as the `activeVideoInput`. As a safeguard, if the new input could not be added, you'll re-add the old input.
6. With the configuration complete, call `commitConfiguration` on the `AVCaptureSession`, which will batch the changes together, resulting in a single, atomic modification to the session.
7. If an error was encountered when you created the new `AVCaptureDeviceInput`, message the delegate providing it with an opportunity to handle the error.

Run the application again. Assuming your iOS device has two cameras, tapping the camera icon in the upper-right corner of the screen will now switch between the front and back cameras.

Configuring Capture Devices

`AVCaptureDevice` provides you with a great deal of control over the cameras on your iOS device. Specifically, it enables you to independently adjust and lock the camera's focus, exposure, and white balance. Focus and exposure can also be set based on a specific point of interest, making it easy to add tap-to-focus and tap-to-expose features to your applications.

`AVCaptureDevice` also enables you to control the device's LED used for the camera's flash and torch.

Whenever you are making modifications to a camera device, it's critical that you first test that the desired modification is supported by the device. Not all cameras, even on a single iOS device, support all available features. For instance, the front-facing camera does not support focus operations, because it's never more than arm's length from the subject, whereas most back-facing cameras support the full range of focus features. Attempting to make an unsupported device modification will throw an exception, causing your app

to crash, so it's essential to always test the change is supported prior to making the modification. For instance, prior to setting the focus mode to auto, you should first test that particular mode is supported as follows:

[Click here to view code image](#)

```
AVCaptureDevice *device = // Active video capture device  
  
if ([device isFocusModeSupported:AVCaptureFocusModeAutoFocus]) {  
    // Perform configuration  
}
```

After you've verified that the desired configuration change is supported, you can perform the actual device configuration. The basic recipe for modifying a capture device involves locking the device for configuration, making the desired modification, and then unlocking the device. For instance, after verifying that a camera supports the autofocus mode, you could configure its focusMode property as follows:

[Click here to view code image](#)

```
AVCaptureDevice *device = // Active video capture device  
if ([device isFocusModeSupported:AVCaptureFocusModeAutoFocus]) {  
    NSError *error;  
    if ([device lockForConfiguration:&error]) {  
        device.focusMode = AVCaptureFocusModeAutoFocus;  
        [device unlockForConfiguration];  
    } else {  
        // handle error  
    }  
}
```

The devices on your Mac, iPhone, or iPad are shared across the system, so AVCaptureDevice requires you to gain an exclusive lock on the device prior to making any changes. Failing to do so will cause an exception to be thrown. It is not required that you immediately release the lock following the configuration change, but failing to relinquish a lock will cause adverse effects for other applications using the same resource, so in almost all cases you should be a good platform citizen and release the lock after your configuration is complete.

Armed with this knowledge, let's move on and look at how to implement Kamera's various device configuration features.

Adjusting Focus and Exposure

Most back-facing cameras on iOS devices support setting focus and exposure based on a given point of interest. The most intuitive interface to make use of this capability is to allow a user to tap a location on the camera interface and automatically focus or expose at that point. You can additionally lock both the focus and exposure at those points to ensure that your user's perfectly composed shot doesn't change. Let's start by implementing tap-to-focus, as shown in [Listing 6.8](#).

Listing 6.8 Tap-to-Focus Implementation

[Click here to view code image](#)

```
- (BOOL)cameraSupportsTapToFocus // 1
{
    return [[self activeCamera] isFocusPointOfInterestSupported];
}

- (void)focusAtPoint:(CGPoint)point // 2
{
    AVCaptureDevice *device = [self activeCamera];

    if (device.isFocusPointOfInterestSupported // 3
&& [device isFocusModeSupported:AVCaptureFocusModeAutoFocus])
    {

        NSError *error;
        if ([device lockForConfiguration:&error]) // 4
            device.focusPointOfInterest = point;
        device.focusMode = AVCaptureFocusModeAutoFocus;
        [device unlockForConfiguration];
    } else {
        [self.delegate
deviceConfigurationFailedWithError:error];
    }
}
```

1. Begin by implementing the `cameraSupportsTapToFocus` method. This is implemented by asking the active camera if it supports a focus point of interest. The client makes use of this method to update the user interface as needed.

2. The `focusAtPoint:` method is passed a `CGPoint` struct. The point passed to this method has been converted from screen coordinates to capture device coordinates inside the `THPreviewView` class you implemented earlier.
3. After getting a pointer to the active capture device, test that it supports a focus point of interest and additionally verify that it supports the autofocus mode. This mode performs a single-scan autofocus and then sets the `focusMode` to `AVCaptureFocusModeLocked`.
4. Lock the device for configuration. If the lock is acquired, set the `focusPointOfInterest` property to the passed-in `CGPoint` and set the focus mode to `AVCaptureFocusModeAutoFocus`. Finally, call `unlockForConfiguration` on the `AVCaptureDevice` to release the lock.

Run the application. Find a particular object on which you'd like to set focus and perform a single tap on it. The application draws a blue box onscreen and the camera focuses at that point. The focus is now locked. Move the camera and find a new point of interest and perform another tap to focus on the new subject. With the tap-to-focus feature working, let's move on and look at tap to expose.

The default exposure mode for most devices is `AVCaptureExposureModeContinuousAutoExposure`, which will continually adjust the exposure appropriately as the scene changes. You can implement a tap-to-expose-and-lock feature similar to what you did with tap-to-focus; however, it's a bit more challenging. Although the `AVCaptureExposureMode` enum defines an `AVCaptureExposureModeAutoExpose` value, no current iOS devices support this value. This means you'll need to get a bit more creative to implement this feature in the same way as you did tap-to-focus. Let's look at how this is implemented in [Listing 6.9](#).

Listing 6.9 Tap-to-Expose Methods

[Click here to view code image](#)

```
- (BOOL)cameraSupportsTapToExpose // 1
{
    return [[self activeCamera]
isExposurePointOfInterestSupported];
```

```

}

// Define KVO context pointer for observing 'adjustingExposure'
device property.
static const NSString *THCameraAdjustingExposureContext;

- (void)exposeAtPoint:(CGPoint)point {

    AVCaptureDevice *device = [self activeCamera];

    AVCaptureExposureMode exposureMode =
        AVCaptureExposureModeContinuousAutoExposure;

    if (device.isExposurePointOfInterestSupported
&& // 2
[device isExposureModeSupported:exposureMode]) {

        NSError *error;
        if ([device lockForConfiguration:&error])
            // 3

        device.exposurePointOfInterest = point;
        device.exposureMode = exposureMode;

        if ([device
isExposureModeSupported:AVCaptureExposureModeLocked]) {
            [device
addObserver:self // 4
forKeyPath:@"adjustingExposure"
options:NSKeyValueObservingOptionNew
context:&THCameraAdjustingExposureContext];
        }

        [device unlockForConfiguration];
    } else {
        [self.delegate
deviceConfigurationFailedWithError:error];
    }
}

- (void)observeValueForKeyPath:(NSString *)keyPath
    withObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {

    if (context == &THCameraAdjustingExposureContext)
        // 5

    AVCaptureDevice *device = (AVCaptureDevice *)object;
}

```

```

        if (!device.isAdjustingExposure
            // 6
            && [device
isExposureModeSupported:AVCaptureExposureModeLocked]) {

            [object
removeObserver:self
            // 7
            forKeyPath:@"adjustingExposure"
            context:&THCameraAdjustingExposureConte:

            dispatch_async(dispatch_get_main_queue(),
            // 8
            NSError *error;
            if ([device lockForConfiguration:&error]) {
                device.exposureMode =
AVCaptureExposureModeLocked;
                [device unlockForConfiguration];
            } else {
                [self.delegate
deviceConfigurationFailedWithError:error];
            }
        });

    }

} else {
    [super observeValueForKeyPath:keyPath
        ofObject:object
        change:change
        context:context];
}
}

```

- 1.** The `cameraSupportsTapToExpose` method is implemented nearly identically to `cameraSupportsTapToFocus`. Simply ask the active device if it supports an exposure point of interest.
- 2.** Perform the device configuration tests to ensure the desired configuration is supported. In this case check to verify that the `AVCaptureExposureModeContinuousAutoExposure` mode is supported.
- 3.** Lock the device for configuration and set the `exposurePointOfInterest` and `exposureMode` properties to their desired values.
- 4.** Verify that the device supports the *locked* exposure mode. If so, use

KVO to determine the state of the device's `adjustingExposure` property. Observing this property will enable you to determine when the exposure adjustment has completed, providing you the opportunity to lock the exposure at that point.

5. Verify that the observation callback is the specific change you are interested in by testing that the context argument is the `&THCameraAdjustingExposureContext` pointer.
6. Determine if the device is *no longer* adjusting the exposure level and verify that the device's `exposureMode` can be set to `AVCaptureExposureModeLocked`. If so...
7. Remove `self` as an observer of the `adjustingExposure` property so you won't get notified of subsequent changes.
8. Finally, asynchronously dispatch back to the main queue, providing it a block in which you'll set the `exposureMode` property to `AVCaptureExposureModeLocked`. It's essential the `exposureMode` change be moved to the next run of the event loop so the `removeObserver:` call you made in #7 has the opportunity to complete.

Yes, this was considerably more complex than implementing the tap-to-focus feature; however, you'll find the net result is very similar. Run the application again. Find some dark area in the preview view and double-tap it. You'll see an orange box drawn onscreen, and the exposure will lock at that point. Point the camera at a bright area (your computer screen works well) and double-tap again; you'll see the exposure adjusted appropriately.

Adding the capability of tapping to set and lock the focus and exposure adds some nice polish to the Kamera app. However, you'll need to provide a way for the user to get back to continuous focus and exposure modes. Let's implement that now. [Listing 6.10](#) shows the implementation for the `resetFocusAndExposureModes` method.

Listing 6.10 Resetting Focus and Exposure

[Click here to view code image](#)

```
- (void)resetFocusAndExposureModes {
    AVCaptureDevice *device = [self activeCamera];
```

```

AVCaptureFocusMode focusMode =
AVCaptureFocusModeContinuousAutoFocus;

BOOL canResetFocus = [device isFocusPointOfInterestSupported]
&& // 1
[device isFocusModeSupported:focusMode];

AVCaptureExposureMode exposureMode =
AVCaptureExposureModeContinuousAutoExposure;

BOOL canResetExposure = [device
isExposurePointOfInterestSupported] && // 2
[device
isExposureModeSupported:exposureMode];

CGPoint centerPoint = CGPointMake(0.5f,
0.5f); // 3

NSError *error;
if ([device lockForConfiguration:&error]) {

    if (canResetFocus) // 4
        device.focusMode = focusMode;
        device.focusPointOfInterest = centerPoint;
    }

    if (canResetExposure) // 5
        device.exposureMode = exposureMode;
        device.exposurePointOfInterest = centerPoint;
    }

    [device unlockForConfiguration];
}

} else {
    [self.delegate deviceConfigurationFailedWithError:error];
}

}

```

- 1.** First verify that the focus point of interest and continuous auto focus modes are supported.
- 2.** Perform a similar test to ensure the exposure can be reset using its related feature tests.
- 3.** Create a CGPoint with x and y values of 0.5f. Recall that capture device space is normalized to 0,0 in the upper left and 1,1 in the bottom

right. Creating a `CGPoint` at `0.5,0.5` will perform the initial scan in the middle of the camera space.

4. Lock the device for configuration and, if the focus can be reset, you'll make the desired modifications.
5. Likewise, if the exposure can be set, set the exposure mode as desired.

Run the application. Make some focus and exposure adjustments. Performing a two-finger double-tap anywhere on the preview view will cause the focus and exposure to be reset to their continuous modes.

Adjusting the Flash and Torch Modes

AVCaptureDevice enables you to modify the camera's flash and torch modes. The flash LED on the back of the device is used for flash purposes when taking still images or when used as a continuous light (torch) when working with video. The capture device's `flashMode` and `torchMode` properties can be set to one of three values:

- **AVCapture(Torch|Flash)ModeOn**: Always on.
- **AVCapture(Flash|Torch)ModeOff**: Always off.
- **AVCapture(Flash|Torch)ModeAuto**: The system will automatically turn the LED on and off based on ambient lighting conditions.

Let's look at the implementation of the flash and torch mode methods in [Listing 6.11](#).

Listing 6.11 Flash and Torch Methods

[Click here to view code image](#)

```
- (BOOL)cameraHasFlash {
    return [[self activeCamera] hasFlash];
}

- (AVCaptureFlashMode)flashMode {
    return [[self activeCamera] flashMode];
}

- (void)setFlashMode:(AVCaptureFlashMode)flashMode {
    AVCaptureDevice *device = [self activeCamera];
```

```

        if ([device isFlashModeSupported:flashMode]) {

            NSError *error;
            if ([device lockForConfiguration:&error]) {
                device.flashMode = flashMode;
                [device unlockForConfiguration];
            } else {
                [self.delegate
                deviceConfigurationFailedWithError:error];
            }
        }

    }

- (BOOL)cameraHasTorch {
    return [[self activeCamera] hasTorch];
}

- (AVCaptureTorchMode)torchMode {
    return [[self activeCamera] torchMode];
}

- (void)setTorchMode:(AVCaptureTorchMode)torchMode {

    AVCaptureDevice *device = [self activeCamera];

    if ([device isTorchModeSupported:torchMode]) {

        NSError *error;
        if ([device lockForConfiguration:&error]) {
            device.torchMode = torchMode;
            [device unlockForConfiguration];
        } else {
            [self.delegate
            deviceConfigurationFailedWithError:error];
        }
    }
}

```

I won't provide a full step-by-step walkthrough of this code, because this should look quite familiar to you by now. As always, you need to verify that the particular feature you are attempting to modify is supported prior to making the configuration change.

The Kamera app's main view controller will correctly invoke the `setFlashMode:` or `setTorchMode:` method, depending on the position of the Video/Photo mode selector on the user interface. Run the application.

Tapping the lightning bolt icon in the upper-left corner of the screen will allow you to toggle between the various modes.

Capturing Still Images

When you implemented the `setupSession:` method, you added an instance of `AVCaptureStillImageOutput` to the capture session. This class is a subclass of `AVCaptureOutput` used to capture still images. The `AVCaptureStillImageOutput` class defines the `captureStillImageAsynchronouslyFromConnection:completionHandler:` method to perform the actual capture. Let's look at a simple example.

[Click here to view code image](#)

```
AVCaptureConnection *connection = // Active video capture
connection
id completionHandler = ^(CMSampleBufferRef buffer, NSError *error)
{
    // Handle image capture
};
[imageOutput
captureStillImageAsynchronouslyFromConnection:connection
completionHandler:complet:
```

This method introduces a couple new object types to discuss; the first is `AVCaptureConnection`. When you create a session and add capture device inputs and capture outputs to it, the session automatically forms connections between them, routing the signal flow as needed. Having access to these connections is useful in a number of cases because it provides you with greater control over the data being sent to the outputs. The other new object type is `CMSampleBuffer`. A `CMSampleBuffer` is a Core Foundation object provided by the Core Media framework. We'll spend a fair amount of time examining the details of this object type in the next chapter, but for the time being you can be aware that this object holds the captured image data. Because you specified `AVVideoCodecJPEG` as the codec key when you created the still image output object, the bytes contained within this object will be in compressed JPEG format.

Let's see how these objects are used when capturing still images in the Kamera app (see [Listing 6.12](#)).

Listing 6.12 Capturing Still Images

[Click here to view code image](#)

```
- (void)captureStillImage {
    AVCaptureConnection *connection
=                                     // 1
    [self.imageOutput
connectionWithMediaType:AVMediaTypeVideo];

    if (connection.isVideoOrientationSupported)
{                               // 2
        connection.videoOrientation = [self
currentVideoOrientation];
    }

    id handler = ^(CMSampleBufferRef sampleBuffer, NSError *error)
{
    if (sampleBuffer != NULL) {

        NSData *imageData
=                                     // 4
        [AVCaptureStillImageOutput
jpegStillImageNSDataRepresentation:sampleBuffe:

            UIImage *image = [[UIImage alloc]
initWithData:imageData];           // 5

    } else {
        NSLog(@"NULL sampleBuffer: %@", [error
localizedDescription]);
    }
};

// Capture still
image                                     // 6
[self.imageOutput
captureStillImageAsynchronouslyFromConnection:connection
                                         completionHandle:
}

- (AVCaptureVideoOrientation)currentVideoOrientation {

    AVCaptureVideoOrientation orientation;

    switch ([UIDevice currentDevice].orientation)
{                               // 3
    case UIDeviceOrientationPortrait:
        orientation = AVCaptureVideoOrientationPortrait;
        break;
    case UIDeviceOrientationLandscapeRight:
        orientation = AVCaptureVideoOrientationLandscapeLeft;
        break;
    case UIDeviceOrientationPortraitUpsideDown:
```

```

        orientation =
AVCaptureVideoOrientationPortraitUpsideDown;
        break;
    default:
        orientation = AVCaptureVideoOrientationLandscapeRight;
        break;
    }

    return orientation;
}

```

- 1.** Begin by getting a pointer to the current `AVCaptureConnection` used by the `AVCaptureStillImageOutput` object by calling its `connectionWithMediaType:` method. You will always pass a media type of `AVMediaTypeVideo` when looking up the connection for an `AVCaptureStillImageOutput`.
- 2.** The Kamera app itself supports only portrait orientation, so as you rotate the device, the user interface stays fixed. However, you want to make sure you respect the orientation in which the user is holding the camera, so you can orient the resulting image appropriately. Verify that the connection supports setting its video orientation; if so, set it to the `AVCaptureVideoOrientation` returned by the `currentVideoOrientation` method.
- 3.** Ask the `UIDevice` for its `orientation`, switching over the value to determine the appropriate `AVCaptureVideoOrientation`. It's important to note that left and right `AVCaptureVideoOrientation` values will be *opposite* their `UIDeviceOrientation` counterparts.
- 4.** In the completion handler block, if you received a valid `CMSampleBuffer`, call the `jpegStillImageNSDataRepresentation` class method on the `AVCaptureStillImageOutput` class, which will return an `NSData` representation of the image bytes.
- 5.** Create a new instance of `UIImage` from the `NSData`.

At this point you've successfully captured an image and created a `UIImage` from it. You could present it somewhere in the user interface or write it out to a location in the application sandbox. However, most users will probably be

expecting any images taken with the Kamera app to be written to Camera Roll inside their Photos app. To add this capability to the app, you'll make use of the Assets Library framework.

Writing to the Assets Library

The Assets Library framework provides you programmatic access to the user's photo and video library managed by the iOS Photos app. This framework plays a regular role in many AV Foundation applications, so it's important to learn to use it effectively.

The central class in this framework is `ALAssetsLibrary`. An instance of this class provides the interface to interact with the user's library. This object provides a number of "write" methods that enable you to write either photos or videos to the library. The first time your application attempts to interact with the library, the user will be prompted with a dialog similar to [Figure 6.10](#).

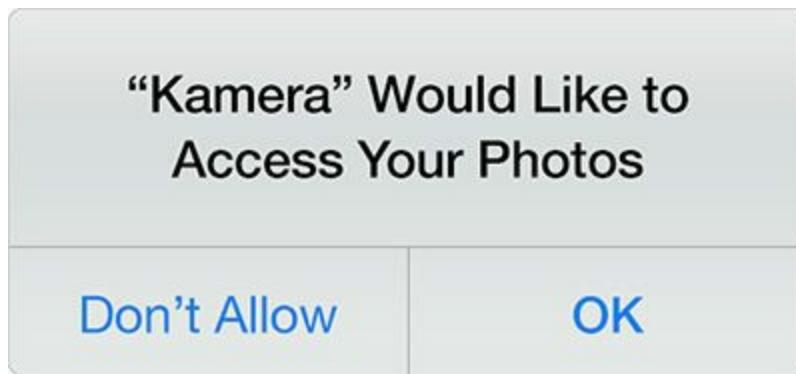


Figure 6.10 Photos access privacy alert

A user must explicitly allow an application access to the library before it can be used. If the user denies access, any attempts to write to the library will fail with an error indicating the user has denied access to your application. If writing to the user's library is critical to the functionality of your application, you should first determine your application's authorization status, as shown in the following example.

[Click here to view code image](#)

```
ALAuthorizationStatus status = [ALAssetsLibrary  
authorizationStatus];  
if (status == ALAuthorizationStatusDenied) {  
    // Show prompt indicating the application won't function  
    // correctly without access to the library
```

```

    } else {
        // Perform authorized access to the library
    }
}

```

Let's look at how to write the captured image to the Assets Library (see [Listing 6.13](#))

Listing 6.13 Capturing a Still Image

[Click here to view code image](#)

```

- (void)captureStillImage {
    AVCaptureConnection *connection =
        [self.imageOutput
    connectionWithMediaType:AVMediaTypeVideo];

    if (connection.isVideoOrientationSupported) {
        connection.videoOrientation = [self
currentVideoOrientation];
    }

    id handler = ^ (CMSampleBufferRef sampleBuffer, NSError *error)
{
    if (sampleBuffer != NULL) {

        NSData *imageData =
            [AVCaptureStillImageOutput
            jpegStillImageNSDataRepresentation:sampleBuffe:

        UIImage *image = [[UIImage alloc]
initWithData:imageData];
[self writeImageToAssetsLibrary:image];                                // 1

        } else {
            NSLog(@"NULL sampleBuffer: %@", [error
localizedDescription]);
        }
    };
    // Capture still image
    [self.imageOutput
captureStillImageAsynchronouslyFromConnection:connection
                                         completionHandle:
}
}

- (void)writeImageToAssetsLibrary:(UIImage *)image {

    ALAssetsLibrary *library = [[ALAssetsLibrary alloc]
init];                                // 2
}

```

```

[library
writeImageToSavedPhotosAlbum:image.CGImage
3
    orientation:
(NSInteger)image.imageOrientation // 4
    completionBlock:^(NSURL *assetURL,
NSError *error) {
    if (!error) {
        [self
postThumbnailNotification:image]; // 5
    } else {
        id message = [error
localizedDescription];
        NSLog(@"Error: %@", message);
    }
}];

- (void)postThumbnailNotification:(UIImage *)image {
    NSNotificationCenter *nc = [NSNotificationCenter
defaultCenter];
    [nc postNotificationName:THThumbnailCreatedNotification
object:image];
}

```

1. In the capture completion handler, call the new method

`writeImageToAssetsLibrary:` passing the `UIImage` created with the image data.

2. Create a new instance of `ALAssetsLibrary` providing you programmatic access to write to the user's Camera Roll.

3. Call the library's

`writeImageToSavedPhotosAlbum:orientation:completionBlock:` method. The `image` argument must be a `CGImageRef`, so you'll ask the `UIImage` for its `CGImage` representation.

4. The orientation argument is an `ALAssetOrientation` enum value. These values correspond directly to the `UIImageOrientation` values returned from the image's `imageOrientation`, so you cast the value as an `NSInteger`.

5. If the write was successful, post a notification with the captured image. This will be used to draw a thumbnail image in the Kamera app's user interface.

Run the application, switch to *Photos* mode, and snap a few photos. The first time you take a photo you'll be prompted with an alert asking for access to the Photos library. Be sure to answer OK. The application provides a simple view of the user's Camera Roll by tapping the image icon to the left of the capture button; however, if you want to see the images in detail, you'll need to switch over to the Photos app.

Still image capture is looking good, so let's move on and look at video capture.

Capturing Videos

The last thing to discuss before wrapping up this chapter is capturing video content. When you set up the capture session, you added an output called `AVCaptureMovieFileOutput`. This class provides a simple and convenient way of capturing QuickTime movies to disk. Most of its core behavior is inherited from its superclass `AVCaptureFileOutput`. This abstract superclass provides a number of useful features, such as the ability to record for a maximum duration or until a particular file size is reached. It can even be configured to ensure a minimum available disk space is observed, which is especially important when recording on a mobile device with limited storage.

Typically when a QuickTime movie is prepared for distribution, the movie header metadata is placed at the beginning of the file. This enables the video player to quickly read the header to determine the content of the file and the structure and location of the various samples it contains. However, when recording a QuickTime movie, the header can't be created until all samples have been captured. After the recording is stopped, the header will be created and appended to the end of the file (see [Figure 6.11](#)).

Distributed QuickTime Movie (Fast Start)



Captured QuickTime Movie



Figure 6.11 Distributed versus captured QuickTime movies

Deferring the creation of the header until all movie samples have been captured can be problematic, especially on a mobile device. If you experience a crash or encounter an interruption, such as getting a phone call, the movie header won't be properly written, leaving you with an unreadable movie file on disk. One of the key features provided by `AVCaptureMovieFileOutput` is its capability to capture QuickTime movies in fragments (see [Figure 6.12](#)).

```
movieFragmentInterval = 10
```

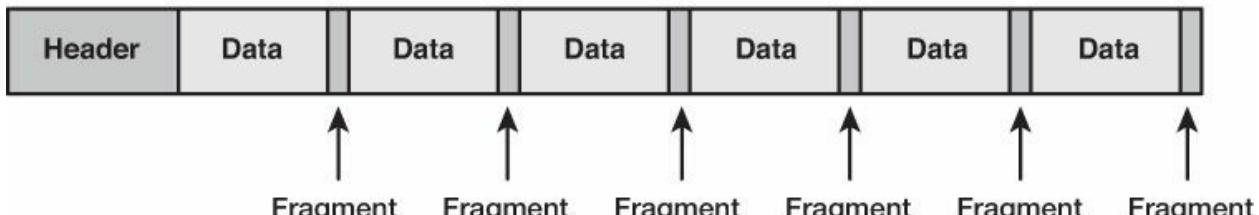


Figure 6.12 Capturing with movie fragments

When a recording begins, a minimal header will be written at the beginning of the file, and as the recording proceeds, a fragment will be written at some periodic interval to build a fully formed header. By default, fragments will be written every 10 seconds, but this value can be changed by modifying the capture output's `movieFragmentInterval` property. Writing fragments enables the QuickTime movie's header to be progressively built, ensuring that in the event of a crash or interruption, the movie will be preserved up to the point of the last written fragment. The default fragment interval will suffice for the Kamera app, but you may want to modify this value based on your own application's needs.

Let's begin implementing the video recording functionality by starting with the transport methods used to start and stop the recording, as shown in [Listing 6.14](#).

Listing 6.14 Video Recording Transport Methods

[Click here to view code image](#)

```
- (BOOL)isRecording
{
    return self.movieOutput.isRecording;
}

- (void)startRecording {
```

```

    if (![self isRecording]) {

        AVCaptureConnection *videoConnection
        = [self.movieOutput
        connectionWithMediaType:AVMediaTypeVideo];

        if ([videoConnection isVideoOrientationSupported])
        {
            // 3
            videoConnection.videoOrientation =
        self.currentVideoOrientation;
        }

        if ([videoConnection isVideoStabilizationSupported])
        {
            // 4
            videoConnection.enablesVideoStabilizationWhenAvailable
        = YES;
        }

        AVCaptureDevice *device = [self activeCamera];

        if (device.isSmoothAutoFocusSupported)
        {
            // 5
            NSError *error;
            if ([device lockForConfiguration:&error]) {
                device.smoothAutoFocusEnabled = YES;
                [device unlockForConfiguration];
            } else {
                [self.delegate
        deviceConfigurationFailedWithError:error];
            }
        }

        self.outputURL = [self
uniqueURL]; // 6
        [self.movieOutput
startRecordingToOutputFileURL:self.outputURL // 8
recordingDelegate:self];

    }

}

- (NSURL *)uniqueURL
{
    NSFileManager *fileManager = [NSFileManager defaultManager];
    NSString *dirPath =
        [fileManager
temporaryDirectoryWithTemplateString:@"kamera.XXXXXX"];

```

```

        if (dirPath) {
            NSString *filePath =
                [dirPath
stringByAppendingPathComponent:@"kamera_movie.mov"];
            return [NSURL fileURLWithPath:filePath];
        }

        return nil;
    }

- (void)stopRecording // 9
{
    if ([self isRecording]) {
        [self.movieOutput stopRecording];
    }
}

```

- 1.** Provide an `isRecording` method that will ask the `AVCaptureMovieFileOutput` instance for its state. This is a support method that exists primarily to provide outside clients some insight on the current state of the controller.
- 2.** Inside the `startRecording` method, get a handle to the current video capture connection. This will be used to configure some key attributes of the video data being captured.
- 3.** Determine if setting the `videoOrientation` property is supported; if so, set it to the current video orientation. Setting the video orientation does not physically rotate the pixel buffers, but is simply used to apply an appropriate transformation matrix on the QuickTime file.
- 4.** If the `enablesVideoStabilizationWhenAvailable` can be set, set it to YES. Not all cameras and devices support this feature, so you need to perform this test. Enabling video stabilization can dramatically improve the quality of video captured, especially on a device like the iPhone. One important point to note is that this impacts only the recorded video. This stabilization effect will not be seen on the video preview screen.
- 5.** A camera has the capability of operating in *smooth focus* mode. This slows the speed at which the camera lens focuses. Normally the camera attempts to autofocus quickly as you're panning the shot, which can lead to a pulsing effect in the captured video. Smoothing the focus

slows the rate at which these focus operations occur, providing a more natural looking video.

6. Look up a unique file system URL where the captured video will be written. You want to maintain a strong reference to this, because its location will be needed later in the processing of the video.
7. The unique URL lookup code should look familiar, because you've used this in the past. This method uses a category method I added to `NSFileManager` called `temporaryDirectoryWithTemplateString`. This creates a uniquely named directory in which the file will be written.
8. Finally, call `startRecordingToOutputFileURL:recordingDelegate:` on the capture output, passing it the `outputURL` and `self` as the delegate. This will begin the actual recording.
9. Add a method to stop the recording, which calls `stopRecording` on the capture output.

If you compile the project, you'll notice a compiler warning indicating the recording delegate you specified, `self`, does not adopt the `AVCaptureFileOutputRecordingDelegate` protocol. Let's make a modification to the controller's class extension, as shown in [Listing 6.15](#).

Listing 6.15 Adopting AVCaptureFileOutputRecordingDelegate

[Click here to view code image](#)

```
@interface THCameraController ()  
<AVCaptureFileOutputRecordingDelegate>  
  
@property (strong, nonatomic) AVCaptureSession *captureSession;  
@property (weak, nonatomic) AVCaptureDeviceInput  
*activeVideoInput;  
  
@property (strong, nonatomic) AVCaptureStillImageOutput  
*imageOutput;  
@property (strong, nonatomic) AVCaptureMovieFileOutput  
*movieOutput;  
@property (strong, nonatomic) NSURL *outputURL;  
  
@end
```

Adopting this protocol means you also need to implement its one required method. You'll use this method to get the finalized file and write it to the Camera Roll. [Listing 6.16](#) provides the implementation for these methods.

Listing 6.16 Writing the Captured Video

[Click here to view code image](#)

```
- (void)captureOutput:(AVCaptureFileOutput *)captureOutput
didFinishRecordingToOutputFileAtURL:(NSURL *)outputFileURL
fromConnections:(NSArray *)connections
error:(NSError *)error {
    if (error)
    {
        [self.delegate mediaCaptureFailedWithError:error];
    } else {
        [self writeVideoToAssetsLibrary:[self.outputURL copy]];
    }
    self.outputURL = nil;
}

- (void)writeVideoToAssetsLibrary:(NSURL *)videoURL {

    ALAssetsLibrary *library = [[ALAssetsLibrary alloc]
init]; // 2

    if ([library
videoAtPathIsCompatibleWithSavedPhotosAlbum:videoURL]) { // 3

        ALAssetsLibraryWriteVideoCompletionBlock completionBlock;

        completionBlock = ^ (NSURL *assetURL, NSError *error)
{
            // 4
            if (error) {
                [self.delegate
assetLibraryWriteFailedWithError:error];
            } else {
                [self generateThumbnailForVideoAtURL:videoURL];
            }
};

        [library
writeVideoAtPathToSavedPhotosAlbum:videoURL // 8
completionBlock:completionBlock];
    }
}

- (void)generateThumbnailForVideoAtURL:(NSURL *)videoURL {
```

```

dispatch_async(self.videoQueue, ^{
    AVAsset *asset = [AVAsset assetWithURL:videoURL];

    AVAssetImageGenerator *imageGenerator
    = [AVAssetImageGenerator
      assetImageGeneratorWithAsset:asset];
    imageGenerator.maximumSize = CGSizeMake(100.0f, 0.0f);
    imageGenerator.appliesPreferredTrackTransform = YES;

    CGImageRef imageRef = [imageGenerator
        copyCGImageAtTime:kCMTimeZero // 6
        actualTime:NUL
        error:nil];
    UIImage *image = [UIImage imageWithCGImage:imageRef];
    CGImageRelease(imageRef);

    dispatch_async(dispatch_get_main_queue(),
                  // 7
                  [self postThumbnailNotification:image];
    });
});
}

```

- 1.** In the delegate callback, if an error was returned you simply message the delegate, providing it an opportunity to handle the error. If no error was encountered, you attempt to write the video to the user's Camera Roll by calling the `writeVideoToAssetsLibrary:` method.
- 2.** Create an `ALAssetsLibrary` instance, which will provide you the interface to write the video.
- 3.** Prior to attempting to write to the assets library, you should call its `videoAtPathIsCompatibleWithSavedPhotosAlbum:` to verify that the video can be written. In this case it should always return `YES`, but this is a good habit to be in when working with the assets library.
- 4.** Create a completion handler block that will be invoked when the write to the assets library finishes. If an error was encountered in this operation, call the delegate to inform it of the error; otherwise, a successful write occurred, so you generate a thumbnail image for the video that will be shown in the user interface.

- 5.** On the `videoQueue`, create a new `AVAsset` and `AVAssetImageGenerator` for the newly created video. Set its `maximumSize` property to a width of 100 and a height of 0, which will size the height of the image according to the video's aspect ratio. You also need to set its `appliesPreferredTrackTransform` to YES so the video's transform (its video orientation) is respected when capturing the thumbnail. Failing to set this will result in an incorrectly oriented thumbnail being created.
- 6.** Because you need to capture only a single image, you use the `copyCGImageAtTime:actualTime:error:` method. This is a synchronous method, which is why you move this operation off the main thread. From the `CGImageRef` returned, you create a `UIImage` that will be shown in the user interface. When calling the `copyCGImageAt-Time:actualTime:error:` method, it's your responsibility to release the created image, so you need to call `CGImageRelease(imageRef)` so that you don't leak memory.
- 7.** Dispatch back to the main thread and post a notification passing the newly created `UIImage`.
- 8.** Perform the actual write to the asset's library, passing the `videoURL` and the `completionBlock`.

Run the application. Record a few seconds of video. Press stop and you should see a new thumbnail created for the scene you just shot. You can tap the thumbnail to open up the browser and see your newly created video.

Summary

You now should have a good understanding of the core AV Foundation capture APIs. You saw how to configure and control an `AVCaptureSession`, saw several examples of how to directly control and manipulate capture devices, and also saw how to use subclasses of `AVCaptureOutput` to capture both still images and videos. Using these core features, you've built an app that provides features and functionality similar to Apple's built-in Camera app. Although this chapter's focus was on iOS, these topics will be used when you're building camera and video apps for that Mac as well. You're off to a great start with AV Foundation's capture APIs, and in the next chapter we'll explore even more advanced capture

features to take your camera and video apps to the next level.

Challenge

Build a simple camera app for the Mac. Almost all Macs have a built-in camera just waiting for some capture code to bring it to life. Much of the code you wrote for the `THCameraController` object can be reused on the Mac, so begin with this class and make modifications as needed.

7. Using Advanced Capture Features

In the previous chapter, you got your first look at AV Foundation's capture classes and gained a good understanding of how to put them into action. Having a mastery of these core capabilities is essential because they provide the foundation on which every photo or video capture application is built. However, AV Foundation still has quite a few tricks up its sleeve. The most recent iOS releases have introduced some amazing new capabilities, and in this chapter, we'll go beyond the basics and dive into the framework's advanced capture features. Let's begin!

Video Zooming

Prior to iOS 7, Apple provided some limited support for camera zooming via the `videoScaleAndCropFactor` property on `AVCaptureConnection`. This enables you to adjust the connection's scaling from its default value of 1.0 up to a maximum determined by its `videoMaxScaleAndCropFactor` property. This feature can be useful in certain cases, but it comes with a couple significant limitations. The first is that `videoScaleAndCropFactor` is a connection-level setting, which means to have this state properly reflected in the user interface, you'll have to apply an appropriate scaling transform to the `AVCaptureVideoPreviewLayer`. This is additional work that needs to be performed and results in reduced image quality in the preview layer when higher scaling factors are applied. The second and larger limitation is that this property can be set only on connections to an `AVCaptureStillImageOutput`, which prevents video capture apps from taking advantage of this feature. Fortunately, a better way to perform zooming was introduced in iOS 7. This feature enables you to apply a zoom factor directly to an `AVCaptureDevice`, which means that all the session's outputs, including the preview layer, will automatically reflect the state of this setting. Let's look at how to use this new feature.

`AVCaptureDevice` provides a property called `videoZoomFactor` that enables you to control the zoom level of the capture device. The minimum value for this property is 1.0, which is the un-zoomed image. The maximum value for this property can be determined by querying capture device's

`activeFormat`, which is an instance of a new type called `AVCaptureDeviceFormat`. This class provides a variety of details about the capabilities of the active capture format, including its `videoMaxZoomFactor`.

The device performs its zooming effect by center cropping the image captured by the camera sensor. When set to a low zoom factor, generally below 1.5, the image is equal to or larger than the output size. This provides the capability of performing modest amounts of zooming without upscaling, which means the full image quality is preserved. The point at which upscaling must begin can be determined by asking the active `AVCaptureDeviceFormat` for its `videoZoomFactorUpscaleThreshold` value.

Let's jump into the sample code and see how you can put this feature to use. You'll find a starter project in the Chapter 7 directory called **ZoomKamera_Starter** (see [Figure 7.1](#)).

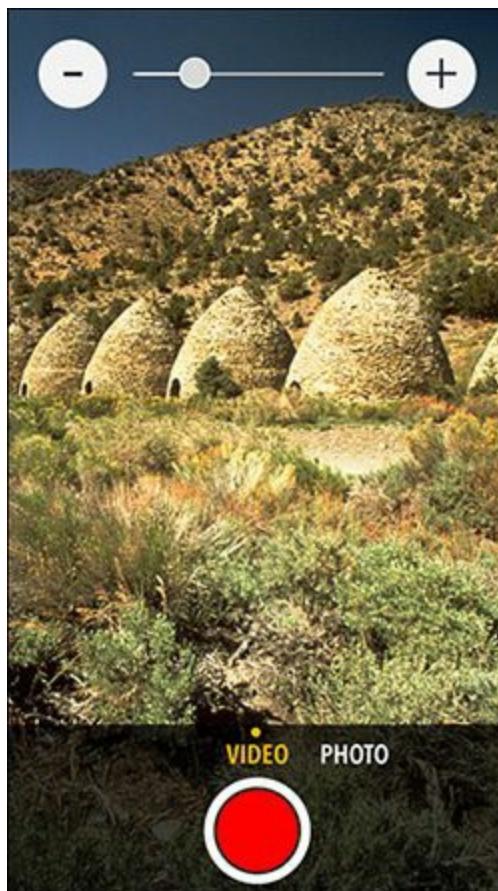


Figure 7.1 Zoom Kamera

Note

The sample projects in this chapter make use of a base class called `THBaseCameraController`. This class is essentially the camera controller you built in [Chapter 6, “Capturing Media,”](#) but with some extension points added. Specifically, it provides methods that can be overridden to configure the session inputs, outputs, and session preset. This will enable us to focus on the new features being introduced without being sidetracked with the topics that we covered in the previous chapter.

In the starter project, you’ll find a stubbed implementation of a class called `THCameraController`. This class extends `THBaseCameraController` and adds the capability of performing video zooming. [Listing 7.1](#) provides the interface for this class.

Listing 7.1 THCameraController Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THCameraZoomingDelegate
<NSObject>                                // 1
- (void)rampedZoomToValue:(CGFloat)value;
@end

@interface THCameraController : THBaseCameraController

@property (weak, nonatomic) id<THCameraZoomingDelegate>
zoomingDelegate;

-
(BOOL)cameraSupportsZoom;                      2
-
- (void)setZoomValue:(CGFloat)zoomValue;
- (void)rampZoomToValue:(CGFloat)zoomValue;
- (void)cancelZoom;

@end
```

1. The `THCameraZoomingDelegate` provides a

`rampedZoomToValue`: method that will be used by the user interface to keep the zoom slider control in sync with the current zoom state.

2. Because not all hardware supports this feature, you'll add a `cameraSupportsZoom` method that the client can query to determine if zoom controls should be displayed.

Let's jump over to the class implementation and begin building out the zooming behavior as shown in [Listing 7.2](#).

Listing 7.2 THCameraController Implementation

[Click here to view code image](#)

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

const CGFloat THZoomRate = 1.0f;

// KVO Contexts
static const NSString *THRampingVideoZoomContext;
static const NSString *THRampingVideoZoomFactorContext;

@implementation THCameraController

- (BOOL)cameraSupportsZoom {
    return self.activeCamera.activeFormat.videoMaxZoomFactor >
1.0f;           // 1
}

- (CGFloat)maxZoomFactor {
    return MIN(self.activeCamera.activeFormat.videoMaxZoomFactor,
4.0f);      // 2
}

- (void)setZoomValue:(CGFloat)zoomValue
{                           // 3
    if (!self.activeCamera.isRampingVideoZoom) {

        NSError *error;
        if ([self.activeCamera lockForConfiguration:&error])
{                           // 4

            // Provide linear feel to zoom slider
            CGFloat zoomFactor = pow([self maxZoomFactor],
zoomValue);          // 5
            self.activeCamera.videoZoomFactor = zoomFactor;
    }
}
```

```

        [self.activeCamera
unlockForConfiguration]; // 6

    } else {
        [self.delegate
deviceConfigurationFailedWithError:error];
    }
}

...
@end

```

- 1.** Begin with the implementation of the `cameraSupportsZoom` method. Ask the `activeCamera`, the currently selected `AVCaptureDevice`, for its active `AVCaptureDeviceFormat`. The capture device supports zooming if the format's `videoMaxZoomFactor` value is greater than `1.0`.
- 2.** To determine the maximum allowed zoom factor, find the `MIN` value of the active format's `videoMaxZoomFactor` and `4.0f`. The value of `4.0f` is arbitrary, but provides a reasonable zooming range. You typically won't want to allow zooming up to the `videoMaxZoomFactor`, because it's usually far greater than is usable, but you should always consider it when calculating the max value. An exception will be raised if you attempt to set a zooming factor beyond its allowed maximum value.
- 3.** The `setZoomValue:` method will be called when the user moves the zoom slider depicted in [Figure 7.1](#). The slider's range is from `0.0f` to `1.0f`.
- 4.** If the capture device is not currently ramping the video zoom, begin by locking the device for configuration. This configuration, like all others, will throw an exception if a lock isn't obtained prior to making the configuration change.
- 5.** The app provides a zoom range of 1x to 4x. This growth is exponential, so to provide a linear feel throughout its range, you calculate a `zoomFactor` value by raising the max zoom factor to the power of the `zoomValue` (0 to 1). After it is calculated, you set the value for the

capture device's `videoZoomFactor` property.

6. Finally, with the configuration complete, you relinquish the lock on the device.

Run the application and move the slider. You'll see that the zoom level is constantly reflected in the preview layer as well as in any videos or photos you capture. Feel free to adjust the upper bound returned from the `maxZoomFactor` method to see the effect of setting higher zoom levels.

The `videoZoomFactor` property makes an immediate adjustment to the zoom level. This works well for a continuous type of control, such as a slider; however, you can also adjust the zoom level by ramping from one value to another over time. This can be useful if you want to build a jog shuttle type of control. The app provides minus (-) and plus (+) buttons at the ends of the slider. Let's add some behavior to these buttons to enable you to adjust the zoom level over time (see [Listing 7.3](#)).

Listing 7.3 Enabling Zoom Ramping

[Click here to view code image](#)

```
- (void)rampZoomToValue:(CGFloat)zoomValue           // 1
{
    CGFloat zoomFactor = pow([self maxZoomFactor], zoomValue);
    NSError *error;
    if ([self.activeCamera lockForConfiguration:&error]) {
        [self.activeCamera
rampToVideoZoomFactor:zoomFactor                  // 2
                           withRate:THZoomRate];
        [self.activeCamera unlockForConfiguration];
    } else {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
}

- (void)cancelZoom                                // 3
{
    NSError *error;
    if ([self.activeCamera lockForConfiguration:&error]) {
        [self.activeCamera cancelVideoZoomRamp];
        [self.activeCamera unlockForConfiguration];
    } else {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
}
```

-
1. Tapping down on the plus or minus button in the user interface calls the `rampZoomToValue:` method. If the user pressed the minus (-) button, the value passed to this method will be `0.0f` and if the user pressed the plus (+) button, the value will be `1.0f`. As you did previously, you calculate an appropriate `zoomFactor` based on `zoomValue` argument.
 2. Call the `rampToVideoZoomFactor:withRate:` method passing the calculated `zoomFactor` and a constant `THZoomRate`, which is equal to `1.0f`. This has the effect of doubling the zoom factor every second. The appropriate rate value is up to you, but this will typically be in the range of `1.0f` to `3.0f` to provide a comfortable feel to the zooming control.
 3. When the user performs a *touch up inside* on the button, the `cancelZoom` method will be invoked. This cancels the current zoom ramp and sets the `zoomFactor` at its current state. As with all device configuration changes, you need to lock the device for configuration, make the desired change, and then unlock when complete.

Run the application again and try this new behavior. You'll see the plus and minus buttons behave as you'd expect and smoothly ramp the video zoom; however, you'll notice a problem in the user interface in that the slider is not reflecting the current zoom level. Let's resolve this problem in [Listing 7.4](#).

Listing 7.4 Adding Zoom State Observers

[Click here to view code image](#)

```
- (BOOL)setupSessionInputs:(NSError ***)error {
    BOOL success = [super
setupSessionInputs:error];
                                // 1
    if (success) {
        [self.activeCamera addObserver:self
                               forKeyPath:@"videoZoomFactor"
                               options:0
                               context:&THRampingVideoZoomFactorContext];
        [self.activeCamera addObserver:self
                               forKeyPath:@"rampingVideoZoom"
                               options:0
                               context:&THRampingVideoZoomContext];
    }
}
```

```

        return success;
    }

- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {
    if (context == &THRampingVideoZoomContext) {
        [self
        updateZoomingDelegate]; // 2
    } else if (context == &THRampingVideoZoomFactorContext) {
        if (self.activeCamera.isRampingVideoZoom) {
            [self
            updateZoomingDelegate];
        } // 3
    } else {
        [super observeValueForKeyPath:keyPath
            ofObject:object
            change:change
            context:context];
    }
}

- (void)updateZoomingDelegate {
    CGFloat curZoomFactor = self.activeCamera.videoZoomFactor;
    CGFloat maxZoomFactor = [self maxZoomFactor];
    CGFloat value = log(curZoomFactor) /
log(maxZoomFactor); // 4
    [self.zoomingDelegate
    rampedZoomToValue:value];
} // 5

```

- 1.** Override the superclass implementation of `setupSessionInputs`: so you can access the `activeCamera` as the capture session is being set up. You add `self` as an observer of the capture device's `videoZoomFactor` and `rampingVideoZoom` key paths.
- 2.** If the context is `&THRampingVideoZoomContext`, call the `updateZoomingDelegate` method. This observation occurs when the ramp begins and again when it ends.
- 3.** If the context is `&THRampingVideoZoomFactorContext` and an active video ramp is in progress, you again call the `updateZoomingDelegate` method.

4. To convert the current zoom level back to the slider's 0 to 1 scale, take the log of the current `videoZoomFactor` divided by the log of the `maxZoomFactor`.
5. Finally, pass this value to the `zoomingDelegate` so the user interface can be updated appropriately.

Now when you run the application again, you'll see the slider value properly update its position as you interact with the jog shuttle controls.

AV Foundation's new zooming capabilities are a great addition to the platform, offering significant benefits over using capture connection's `videoScaleAndCropFactor`. Setting the `videoZoomFactor` directly on the `AVCaptureDevice` means you now have one zoom to rule them all.

Face Detection

If you've used the built-in iOS Camera app, you've probably noticed how it automatically focuses on new faces as they enter the camera's field of view. A yellow box is drawn around a newly detected face, and it performs an autofocus operation at the center point of the rectangle. This is a useful feature because it makes it much easier to quickly capture a focused shot without any manual user interaction. Fortunately, you can implement this same functionality in your own applications by taking advantage of AV Foundation's real-time face detection capability.

Apple's first foray into providing iOS developers face detection capabilities came by way of the Core Image framework. Core Image provides the `CIDetector` and `CIFaceFeature` objects, which are easy to use and provide some rather powerful face detection capabilities. However, these features aren't optimized for real-time use, making it impossible to get the frame rates required in modern camera and video apps. In iOS 6, Apple introduced a new hardware-accelerated feature directly into AV Foundation, enabling the capability of detecting up to ten faces in real-time. This feature is made possible through a specialized `AVCaptureOutput` type called `AVCaptureMetadataOutput`. This output is similar to the other ones you've seen so far, but instead of outputting a still image or a QuickTime movie, it outputs metadata. This metadata comes in the form of an abstract class called `AVMetadataObject`, which defines the interface suitable for handling a variety of metadata types. When you're working with face

detection, it outputs a concrete subclass type called `AVMetadataFaceObject`.

An instance of `AVMetadataFaceObject` provides several properties describing the detected face, the most important of which is the face's bounds. This is a `CGRect` given in the device's scalar coordinates. Recall from [Chapter 6](#) that the device's scalar coordinates range from 0,0 in the upper-left corner to 1,1 in the bottom-right corner in the camera's native orientation.

In addition to the bounds, an `AVMetadataFaceObject` instance provides properties defining the *roll* and *yaw* angles of a detected face. The roll angle indicates the side-to-side tilt of a person's head toward his shoulders, and the yaw angle indicates the face's rotation around the y-axis. Let's build an application to see how you can put this feature to use. You'll find a sample project in the [Chapter 7](#) directory called **FaceKamera_Starter**. We'll begin by building the application's camera controller class as shown in [Listing 7.5](#).

Listing 7.5 THCameraController Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THFaceDetectionDelegate <NSObject>
- (void)didDetectFaces:(NSArray *)faces;
@end

@interface THCameraController : THBaseCameraController

@property (weak, nonatomic) id <THFaceDetectionDelegate>
faceDetectionDelegate;

@end
```

This class, like the others in this chapter, extends from `THBaseCameraController`, which defines the core camera behavior. You'll define a new protocol in this interface called `THFaceDetectionDelegate` and define a property for it in the `THCameraController` class. This delegate will be notified as the camera

captures new metadata. Switch over to the class implementation and begin building out this class's behavior as shown in [Listing 7.6](#).

Listing 7.6 **THCameraController** Implementation

[Click here to view code image](#)

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController ()
@property (strong, nonatomic) AVCaptureMetadataOutput
*metadataOutput;           // 1
@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError ***)error {

    self.metadataOutput = [[AVCaptureMetadataOutput alloc]
init];                      // 2

    if ([self.captureSession canAddOutput:self.metadataOutput]) {
        [self.captureSession addOutput:self.metadataOutput];

        NSArray *metadataObjectTypes =
@+[AVMetadataObjectTypeFace];           // 3
        self.metadataOutput.metadataObjectTypes =
metadataObjectTypes;

        dispatch_queue_t mainQueue = dispatch_get_main_queue();
        [self.metadataOutput
setMetadataObjectsDelegate:self           // 4
queue:mainQueue];
    }

    return YES;
}

} else
{
5
    if (error) {
        NSDictionary *userInfo = @{@"NSLocalizedStringKey":
@"Failed to still image
output."};
        *error = [NSError errorWithDomain:THCameraErrorDomain
code:THCameraErrorFailedT
userInfo:userInfo];
    }
    return NO;
},
```

```
    }  
}  
  
@end
```

1. Begin by creating a property inside the class extension to store the `AVCaptureMetadataOutput` instance used by this class.
2. Completely override the `setupSessionOutputs` : method because the superclass implementation of this method won't be needed in this case. Instead, create a new instance of `AVCaptureMetadataOutput` and add it as the capture session's output.
3. When configuring an `AVCaptureMetadataOutput` object, it's important to specify what types of metadata this object should output by setting its `metadataObjectTypes` property. Limiting the set of metadata types it detects acts as a performance optimization and reduces the objects to just those that are of interest to you. AV Foundation supports a number of metadata types, but because you're currently interested only in *face* metadata, you provide it an `NSArray` containing a single element with the constant value `AVMetadataObjectTypeFace`.
4. An `AVCaptureMetadataOutput` requires a delegate object that will be called as new metadata is detected. You mark `self` as the delegate and you provide a serial dispatch queue on which its callbacks will occur. This could be a custom serial dispatch queue; however, because the face detection is hardware accelerated and the interesting work to be done will be on the main queue, you specify the main queue for this argument.
5. Finally, if the output couldn't be added, you populate the `error` pointer with an appropriate `NSError` and return `NO`.

If you build the project, you'll see a compiler warning indicating that `self` isn't a valid delegate because it doesn't adopt the `AVCaptureMetadataOutputObjectsDelegate` protocol. Before the controller can be notified when new metadata is detected, it needs to adopt this protocol and implement its `captureOutput:didOutputMetadataObjects:fromConnection:`

method. Let's resolve this issue by making the following additions to the THCameraController class, as shown in [Listing 7.7](#).

Listing 7.7 Adopting AVCaptureMetadataOutputObjectsDelegate

[Click here to view code image](#)

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController : NSObject
<AVCaptureMetadataOutputObjectsDelegate>           // 1
...
@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError ***)error {
...
}

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputMetadataObjects:(NSArray *)metadataObjects
fromConnection:(AVCaptureConnection *)connection {
    for (AVMetadataFaceObject *face in metadataObjects)
    {
        NSLog(@"Face detected with ID: %li", (long)face.faceID);
        NSLog(@"Face bounds: %@", NSStringFromCGRect(face.bounds));
    }
    [self.faceDetectionDelegate
    didDetectFaces:metadataObjects];           // 3
}
}

@end
```

- 1.** Make the class extension adopt the `AVCaptureMetadataOutputObjectsDelegate` protocol.
- 2.** In the implementation of the protocol's method, add some temporary debugging code to verify that the detection is occurring as expected. Iterate through each detected face, and for each output its `faceID` and its `bounds`. This debugging code can be removed after you've verified

the configuration is complete.

3. Finally, call the delegate object's `didDetectFaces:` method, passing it the array of `AVMetadataObject` instances output by the `AVCaptureMetadataOutput` instance.

Run the application. As a face enters the camera's field of view, you'll see the debug messages being sent to the console. Remember, the bounds will be given in device coordinates. In a moment, you'll see how these can be converted to a more usable coordinate space.

Building the Face Detection Delegate

Most of the book's code examples have been factored in such a way that we can largely stay out of the user interface layer. However, working with face detection is one case where we'll need to step into some UIKit and Core Animation code to get a better sense for how this metadata can be used. The object adopting the `THFaceDetectionDelegate` protocol in this application is the `THPreviewView` class. Let's begin by looking at its interface in [Listing 7.8](#).

Listing 7.8 `THPreviewView` Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>
#import "THFaceDetectionDelegate.h"

@interface THPreviewView : UIView <THFaceDetectionDelegate>

@property (strong, nonatomic) AVCaptureSession *session;

@end
```

This class has a simple interface; the most interesting point to note is that it adopts the `THFaceDetectionDelegate` protocol. This class will be the target of the metadata and will be used to provide a visual representation of it. Switch over to the implementation and let's begin building this class. There is a fair amount of code in the view implementation, so you'll build this out in pieces starting with the basic class structure, as shown in [Listing 7.9](#).

Listing 7.9 `THPreviewView` Implementation

[Click here to view code image](#)

```
#import "THPreviewView.h"

@interface THPreviewView
()
// 1
@property (strong, nonatomic) CALayer *overlayLayer;
@property (strong, nonatomic) NSMutableDictionary *faceLayers;
@property (nonatomic, readonly) AVCaptureVideoPreviewLayer
*previewLayer;
@end

@implementation THPreviewView

+ (Class)layerClass
{
    return [AVCaptureVideoPreviewLayer class];
// 2
}

- (id)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        [self setupView];
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder];
    if (self) {
        [self setupView];
    }
    return self;
}

- (void)setupView {
    //
}

- (AVCaptureSession*)session {
    return self.previewLayer.session;
}

- (void)setSession:(AVCaptureSession *)session
{
    // 3
    self.previewLayer.session = session;
}

- (AVCaptureVideoPreviewLayer *)previewLayer {
    return (AVCaptureVideoPreviewLayer *)self.layer;
}
```

```
}

- (void)didDetectFaces:(NSArray *)faces {
    //
}

@end
```

1. Create a class extension and define the internal properties used by the class.
2. Override the `layerClass` method and return an instance of `AVCaptureVideoPreviewLayer` as you did in the previous chapter. This technique will automatically make the `AVCaptureVideoPreviewLayer` the *backing* layer for this class when a new instance is created.
3. To make the association between the `AVCaptureSession` and the `AVCapturePreviewLayer`, override the `setSession:` method and set the `AVCaptureSession` instance as the preview layer's session property.

With the basic class structure set up, let's continue and provide an implementation of the `setupView` method in [Listing 7.10](#).

Listing 7.10 Implementing the `setupView` Method

[Click here to view code image](#)

```
@implementation THPreviewView

...

- (void)setupView {
    self.faceLayers = [NSMutableDictionary
dictionary]; // 1
    self.previewLayer.videoGravity =
AVLayerVideoGravityResizeAspectFill;

    self.overlayLayer = [CALayer
layer]; // 2
    self.overlayLayer.frame = self.bounds;
    self.overlayLayer.sublayerTransform =
THMakePerspectiveTransform(1000);
    [self.previewLayer addSublayer:self.overlayLayer];
}
```

```
static CATransform3D THMakePerspectiveTransform(CGFloat  
eyePosition) { // 3  
    CATransform3D transform = CATransform3DIdentity;  
    transform.m34 = -1.0 / eyePosition;  
    return transform;  
}  
  
...  
  
@end
```

1. Initialize the `faceLayers` property with an `NSMutableDictionary` used to store the layer instances corresponding to the detected faces. Additionally, set the `videoGravity` to `AVLayerVideoGravityResizeAspectFill` so it fills the preview layer's bounds. See [Chapter 6](#) for a description of the available video gravity values.
2. Initialize the `overlayLayer` property with a new `CALayer` instance and set its bounds equal to the view's bounds so that it completely fills the preview layer. Set the layer's `sublayerTransform` property to a `CATransform3D` configured to apply a perspective transform to any sublayers.
3. Create a function to return a `CATransform3D` type. This is the *transformation matrix* type used by Core Animation for applying transforms such as scaling and rotation. Setting the `m34` element enables you to apply a perspective transformation, which will enable its sublayers to be rotated around the Y-axis.

If you haven't used Core Animation before, some of this may seem confusing, but don't worry, you'll see the effects of this configuration shortly. I would recommend Nick Lockwood's *iOS Core Animation* (2014, Boston: Addison-Wesley) for a great overview of how to effectively use the Core Animation framework.

The metadata captured by an `AVCaptureMetadataOutput` object is in device space. To use this metadata, you first need to translate the data it provides into the view's coordinate space. Fortunately, this is trivial to do because `AVCaptureVideoPreviewLayer` provides the method needed

to perform the hard work required for this transformation. Listing 7.11 shows how this is done.

Listing 7.11 Transforming Metadata

[Click here to view code image](#)

```
@implementation THPreviewView

...
- (void)detectFaces:(NSArray *)faces {
    NSArray *transformedFaces = [self
        transformedFacesFromFaces:faces]; // 1
    // Process transformed faces. To be implemented in Listing
    // 7.12.
}

- (NSArray *)transformedFacesFromFaces:(NSArray *)faces
{
    NSMutableArray *transformedFaces = [NSMutableArray array];
    for (AVMetadataObject *face in faces) {
        AVMetadataObject *transformedFace
        = [self.previewLayer
            transformedMetadataObjectForMetadataObject:face];
        [transformedFaces addObject:transformedFace];
    }
    return transformedFaces;
}

...
@end
```

1. Create a local NSArray to store the transformed faces.
2. Create a new method to transform the device coordinate-space face objects to a collection of view coordinate-space objects.
3. Iterate through the collection of faces passed to the method, and for each, call the preview layer's `transformedMetadataObjectForMetadataObject:` method and add it to the `transformedFaces` array.

You now have a collection of `AVMetadataFaceObject` instances that

have meaningful coordinates for constructing the user interface. Let's continue with the implementation of the preview view in [Listing 7.12](#).

Listing 7.12 Visualizing the Detected Faces

[Click here to view code image](#)

```
@implementation THPreviewView
...
- (void)detectFaces:(NSArray *)faces {
    NSArray *transformedFaces = [self
        transformedFacesFromFaces:faces];
    NSMutableArray *lostFaces = [self.faceLayers.allKeys
        mutableCopy]; // 1
    for (AVMetadataFaceObject *face in transformedFaces) {
        NSNumber *faceID =
        @(face.faceID); // 2
        [lostFaces removeObject:faceID];
        CALayer *layer =
        self.faceLayers[faceID]; // 3
        if (!layer) {
            // no layer for faceID, create new face layer
            layer = [self
                makeFaceLayer]; // 4
            [self.overlayLayer addSublayer:layer];
            self.faceLayers[faceID] = layer;
        }
        layer.transform =
        CATransform3DIdentity; // 5
        layer.frame = face.bounds;
    }
    for (NSNumber *faceID in lostFaces)
    { // 6
        CALayer *layer = self.faceLayers[faceID];
        [layer removeFromSuperlayer];
        [self.faceLayers removeObjectForKey:faceID];
    }
}
```

```

- (CALayer *)makeFaceLayer
{
    CALayer *layer = [CALayer layer];
    layer.borderWidth = 5.0f;
    layer.borderColor =
        [UIColor colorWithRed:0.188 green:0.517 blue:0.877
alpha:1.000].CGColor;
    return layer;
}

...
@end

```

- 1.** Create a mutable copy of the key values contained in the `faceLayers` dictionary. This array is used to determine which faces have gone out of view and should have their layers removed from the user interface.
- 2.** Iterate through each of the transformed face objects and capture its associated `faceID`, which uniquely identifies a detected face. Remove its entry from the `lostFaces` array so its layer won't be removed from the user interface at the end of this method.
- 3.** Look up a `CALayer` instance from the `faceLayers` dictionary for the current `faceID`. This dictionary acts as a temporary cache of `CALayer` objects.
- 4.** If no layer was found for the given `faceID`, create a new face layer by calling the `makeFaceLayer` method and add it to the `overlayLayer`. Finally, add it to the dictionary so it can be reused on previous invocations of the `didDetectFaces` : method.
- 5.** The `makeFaceLayer` method creates a new `CALayer` with a 5-point light blue border and returns the layer to the caller.
- 6.** Set the layer's `transform` property to `CATransform3DIdentity`. An identity transform is the layer's default transform, which essentially means its untransformed state. This will have the effect of resetting any previously applied transforms that you'll be applying shortly. You also set the layer's frame based on the face object's `bounds` property.
- 7.** Finally, iterate through the collection of remaining face IDs contained

in the `lostFaces` array and remove them from their super layer and the `faceLayers` dictionary.

You're now ready to run the application and see this feature in action. The app is set up with the front-facing camera active by default, so as you look into the camera you'll see a blue box drawn around your face that tracks your movement in real-time. As cool as that is, you're not currently using some of the other interesting data provided by `AVMetadataFaceObject`. We've discussed how the face object also contains data describing the roll and yaw of the detected face. Let's see how this can be depicted in the user interface (see [Listing 7.13](#)).

Listing 7.13 Visualizing Roll and Yaw

[Click here to view code image](#)

```
@implementation THPreviewView

...
- (void)didDetectFaces:(NSArray *)faces {
    NSArray *transformedFaces = [self
        transformedFacesFromFaces:faces];
    NSMutableArray *lostFaces = [self.faceLayers.allKeys
        mutableCopy];
    for (AVMetadataFaceObject *face in transformedFaces) {
        NSNumber *faceID = @(face.faceID);
        [lostFaces removeObject:faceID];
        CALayer *layer = self.faceLayers[faceID];
        if (!layer) {
            // no layer for faceID, create new face layer
            layer = [self makeFaceLayer];
            [self.overlayLayer addSublayer:layer];
            self.faceLayers[faceID] = layer;
        }
        layer.transform =
        CATransform3DIdentity;                                // 1
        layer.frame = face.bounds;
        if (face.hasRollAngle) {
            CATransform3D t = [self
```

```

        transformForRollAngle:face.rollAngle]; // 2
                layer.transform = CATransform3DConcat(layer.transform,
t);
            }

            if (face.hasYawAngle) {
                CATransform3D t = [self
transformForYawAngle:face.yawAngle]; // 4
                layer.transform = CATransform3DConcat(layer.transform,
t);
            }
        }

        for (NSNumber *faceID in lostFaces)
    { // 6
        CALayer *layer = self.faceLayers[faceID];
        [layer removeFromSuperlayer];
        [self.faceLayers removeObjectForKey:faceID];
    }

}

// Rotate around Z-axis
- (CATransform3D)transformForRollAngle:(CGFloat)rollAngleInDegrees
{ // 3
    CGFloat rollAngleInRadians =
    THDegreesToRadians(rollAngleInDegrees);
    return CATransform3DMakeRotation(rollAngleInRadians, 0.0f,
0.0f, 1.0f);
}

// Rotate around Y-axis
- (CATransform3D)transformForYawAngle:(CGFloat)yawAngleInDegrees
{ // 5
    CGFloat yawAngleInRadians =
    THDegreesToRadians(yawAngleInDegrees);

    CATransform3D yawTransform =
        CATransform3DMakeRotation(yawAngleInRadians, 0.0f, -1.0f,
0.0f);

    return CATransform3DConcat(yawTransform, [self
orientationTransform]);
}

- (CATransform3D)orientationTransform
{ // 6
    CGFloat angle = 0.0;
    switch ([UIDevice currentDevice].orientation) {
        case UIDeviceOrientationPortraitUpsideDown:

```

```

        angle = M_PI;
        break;
    case UIDeviceOrientationLandscapeRight:
        angle = -M_PI / 2.0f;
        break;
    case UIDeviceOrientationLandscapeLeft:
        angle = M_PI / 2.0f;
        break;
    default: // as UIDeviceOrientationPortrait
        angle = 0.0;
        break;
    }
    return CATransform3DMakeRotation(angle, 0.0f, 0.0f, 1.0f);
}

static CGFloat THDegreesToRadians(CGFloat degrees) {
    return degrees * M_PI / 180;
}

...
@end

```

- 1.** For each face layer, begin by setting its `transform` property to `CATransform3DIdentity`. This has the effect of resetting any previously applied transforms.
- 2.** Determine if the face object has a valid roll angle by testing its `hasRollAngle` property. If `hasRollAngle` returns NO, asking for its `rollAngle` property will raise an exception. If the face object has a `rollAngle`, ask for an appropriate `CATransform3D`, concatenate it with the identify transform, and set the layer's `transform` property.
- 3.** The `rollAngle` obtained from the face object is in degrees, but you need to convert it to radians as required by the Core Animation transform functions. Pass this converted angle to the `CATransform3DMakeRotation` function along with the values 0, 0, 1 for the X, Y, Z components. This will result in a roll angle rotation transformation around the Z-axis.
- 4.** As you did when calculating the `rollAngle`, you first need to ask if the face has a yaw angle. If so, ask for its value and calculate an appropriate transform, concatenate it with the layer's existing transform, and set it as the layer's `transform` property.

5. Constructing a transform for the yaw angle is similar to what you did when creating the roll angle. You first convert the angle to radians and create a rotation transform, but this time you'll be rotating around the Y-axis. Because the `overlayLayer` has the required `sublayerTransform` applied, the layer will be projected along the Z-axis, resulting in a 3D-effect as the face turns from side-to-side.
6. The application's user interface is fixed in portrait orientation, but you need to calculate an appropriate rotation transform for the device's orientation. Failing to do this will cause the face layer's yaw effect to be drawn incorrectly. This transform will be concatenated with the others.

Run the application again. Now as you tilt your head side-to-side, you'll see the box rotate and track your movements. Likewise, as you swivel your head toward your shoulders, you'll see the box track the yaw and project itself into a 3D space.

You should now have a solid understanding of using AV Foundation face detection. We're really just scratching the surface of the kinds of user interfaces you could build using this capability. I hope thoughts of overlaying hats, glasses, and mustaches have popped into your head! It is entirely possible to capture still images that composite the Core Animation layers with the video images; however, this requires a significant Quartz framework play, which is outside the scope of this book. I would suggest looking at Apple's SquareCam sample available from the Apple Developer Connection site or its StacheCam example from WWDC 2013 for examples of how to perform this compositing.

Machine-Readable Code Detection

A significant new feature provided by AV Foundation is its capability of detecting machine-readable codes, which is the more formal way of saying it supports barcode scanning. The framework provides real-time detection of a broad set of barcode symbologies, works with both the front and back cameras, and is supported by all devices capable of running iOS 7 and iOS 8. Before getting into the technical details of this feature, let's take a quick look at the different barcode symbologies the framework supports.

1D Codes

1D codes are the most prevalent type of barcodes you'll see. These are widely used in shipping, manufacturing, and retail, and they play an important role in most inventory control systems. AV Foundation supports the 1D symbologies shown in [Table 7.1](#).

Code	Example
UPC-E	 0 012345 7
EAN-8	 0123 4565
EAN-13	 0 123456 789104
Code 39 (with and without checksum)	 0123456789
Code 93	 1 INFINITE LOOP

Code 128



Interleaved 2 of 5 (iOS 8 only)



ITF 14 (iOS 8 only)

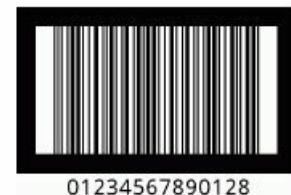


Table 7.1 Supported 1D Symbologies

AV Foundation additionally supports three 2D symbologies. QR Codes are used primarily for mobile marketing, but you'll find many creative uses of these codes in the mobile space. Aztec codes are widely used by the airline industry for boarding passes. PDF417 is commonly used in shipping applications. [Table 7.2](#) provides examples of these codes.

Code	Example
QR	
Aztec	
PDF-417	
Data Matrix (iOS 8 only)	

Table 7.2 **Supported 2D Symbologies**

A discussion of the details of barcodes is outside the scope of this book, but you'll find a nice summary of the features and uses for these symbologies at <http://makebarcode.com/>.

Building a Barvode Scanner

Let's create a sample app that makes use of this feature. You'll find a sample project in the [Chapter 7](#) directory called **CodeKamera_Starter**. We'll begin by building the camera controller object shown in [Listing 7.14](#).

Listing 7.14 **THCameraController** Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>
```

```
#import "THBaseCameraController.h"

@protocol THCodeDetectionDelegate <NSObject>
- (void)didDetectCodes:(NSArray *)codes;
@end

@interface THCameraController : THBaseCameraController

@property (weak, nonatomic) id <THCodeDetectionDelegate>
codeDetectionDelegate;

@end
```

The interface for `THCameraController` is nearly identical to the one you created for the `FaceKamera` app; the only difference is the definition of a new delegate protocol called `THCodeDetectionDelegate`. This delegate defines a single `didDetectCodes:` method that will be called as new barcodes are detected. Let's switch to the implementation and begin building out its behavior (see [Listing 7.15](#)).

Listing 7.15 Camera Setup

[Click here to view code image](#)

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>
@interface THCameraController ()
<AVCaptureMetadataOutputObjectsDelegate> // 1
@property (strong, nonatomic) AVCaptureMetadataOutput
*metadataOutput;
@end

@implementation THCameraController

- (NSString *)sessionPreset
{
    return AVCaptureSessionPreset640x480; // 2
}

- (BOOL)setupSessionInputs:(NSError * __autoreleasing *)error {
    BOOL success = [super setupSessionInputs:error];
    if (success) {
        if (self.activeCamera.autoFocusRangeRestrictionSupported)
{            // 3
            if ([self.activeCamera lockForConfiguration:error]) {
```

```
        self.activeCamera.autoFocusRangeRestriction =
            AVCaptureAutoFocusRangeRestrictionNear,
        [self.activeCamera unlockForConfiguration];
    }
}
return success;
}

...
@end
```

1. Begin by creating a class extension to store a reference to the `AVCaptureMetadataOutput`. As you did in the FaceKamera app, have this class adopt the `AVCaptureMetadataOutputObjectsDelegate` protocol so it can be notified as new metadata is found.
2. Override the `sessionPreset` method to return an alternate session preset type to use. You're free to use whatever capture preset works best for your app, but Apple recommends using the lowest reasonable resolution to improve performance.
3. A capture device's autofocus functionality normally scans for objects at any distance, which is normally the behavior you'll want in a typical photo or video camera app. However, a property was added in iOS 7 enabling you to tailor this behavior by applying a range restriction. Most barcodes you'll scan will be within a few feet so you can improve the detection's responsiveness by reducing the scanning area. Test to see if this feature is supported and if so, set the `autoFocusRangeRestriction` property to `AVCaptureAutoFocusRangeRestrictionNear`.

With the capture device input configuration complete, let's look at the setup of the capture session's output in [Listing 7.16](#).

Listing 7.16 Configuring the Session Outputs

[Click here to view code image](#)

```
@implementation THCameraController
```

```

...
- (BOOL)setupSessionOutputs:(NSError **)error {
    self.metadataOutput = [[AVCaptureMetadataOutput alloc] init];

    if ([self.captureSession canAddOutput:self.metadataOutput]) {
        [self.captureSession addOutput:self.metadataOutput];

        dispatch_queue_t mainQueue = dispatch_get_main_queue();
        [self.metadataOutput setMetadataObjectsDelegate:self
                                         queue:mainQueue];

        NSArray *types =
        @[AVMetadataObjectTypeQRCode, // 1
          AVMetadataObjectTypeAztecCode];

        self.metadataOutput.metadataObjectTypes = types;

    } else {
        NSDictionary *userInfo = @{@"NSLocalizedStringKey":
                                   @"Failed to add metadata
output."};
        *error = [NSError errorWithDomain:THCameraErrorDomain
                                     code:THCameraErrorFailedToAdd(
                                         userInfo:userInfo];
        return NO;
    }

    return YES;
}

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputMetadataObjects:(NSArray *)metadataObjects
fromConnection:(AVCaptureConnection *)connection {

    [self.codeDetectionDelegate
     didDetectCodes:metadataObjects]; // 2
}
@end

```

1. The implementation of the `setupSessionOutputs:` method is implemented nearly identically as the one used by the FaceKamera app. The only difference is the `metadataObjectTypes` used by the `AVCaptureMetadataOutput` object. In this case, you specify you are interested in scanning QR and Aztec codes.

2. In the delegate callback, invoke with `THCodeDetectionDelegate`, passing it the array of metadata objects that were detected.

The `AVMetadataCaptureOutput` objects emitted when working with barcodes will be instances of `AVMetadataMachineReadableCodeObject`. This object defines a `stringValue` property providing the barcode's actual data value and two properties defining the barcode's geometry. The `bounds` property provides an axis-aligned bounding rectangle of the detected code, and the `corners` property provides an `NSArray` of dictionary representations of its corner points. The latter is particularly useful because it enables you to construct a Bezier path that tightly aligns with the code's corner points.

Building the Code Detection Delegate

The `THCodeDetectionDelegate` in this application is the `THPreviewView`. Let's see how this is implemented, starting in [Listing 7.17](#).

Listing 7.17 THPreviewView Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>
#import "THCodeDetectionDelegate.h"

@interface THPreviewView : UIView <THCodeDetectionDelegate>

@property (strong, nonatomic) AVCaptureSession *session;

@end
```

This interface should look familiar; the only notable point is that it adopts the `THCodeDetectionDelegate` protocol enabling it to be the target of the metadata. Let's move on to the implementation in [Listing 7.18](#). Just as we did when discussing the preview view used by the FaceKamera app, we'll tackle this in sections starting with the basic class structure.

Listing 7.18 THPreviewView Implementation

[Click here to view code image](#)

```

#import "THPreviewView.h"

@interface THPreviewView
()
                                         // 1
@property (strong, nonatomic) NSMutableDictionary *codeLayers;
@end

@implementation THPreviewView

+ (Class)layerClass
{
    return [AVCaptureVideoPreviewLayer class];
}                                         // 2

- (id)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        [self setupView];
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder];
    if (self) {
        [self setupView];
    }
    return self;
}

- (void)setupView
{
    _codeLayers = [NSMutableDictionary dictionary];
    self.previewLayer.videoGravity =
AVLayerVideoGravityResizeAspect;
}                                         // 3

- (AVCaptureSession*)session {
    return self.previewLayer.session;
}

- (void)setSession:(AVCaptureSession *)session
{
    self.previewLayer.session = session;
}                                         // 4

- (AVCaptureVideoPreviewLayer *)previewLayer {
    return (AVCaptureVideoPreviewLayer *)self.layer;
}

```

```
- (void)didDetectCodes:(NSArray *)codes {  
}  
}@end
```

1. Create a class extension and define the internal properties used by the class.
2. Override the `layerClass` method and return an instance of `AVCaptureVideoPreviewLayer` as you did in the previous section. Overriding this method makes the view's *backing layer* an instance of `AVCaptureVideoPreviewLayer`.
3. Initialize the `codeLayers` property with an instance of `NSMutableDictionary`. This will be used to store an array of layers representing the detected barcode's geometry. You also set the layer's `videoGravity` to `AVLayerVideoGravityResizeAspect` so the aspect ratio will be preserved within its bounds.
4. To make the association between the `AVCaptureSession` and the `AVCapturePreviewLayer`, override the `setSession:` method and set the `AVCaptureSession` instance as the preview layer's `session` property.

With the basic class structure set up, let's continue on and start implementing the `didDetectCodes:` method in [Listing 7.19](#). The first step you'll need to take is to transform the metadata from device coordinates to view coordinates.

Listing 7.19 Transforming Metadata

[Click here to view code image](#)

```
@implementation THPreviewView  
...  
- (void)didDetectCodes:(NSArray *)codes {  
    NSArray *transformedCodes = [self  
        transformedCodesFromCodes:codes]; // 1
```

```

}

- (NSArray *)transformedCodesFromCodes:(NSArray *)codes
{
    // 2
    NSMutableArray *transformedCodes = [NSMutableArray array];
    for (AVMetadataObject *code in codes) {
        AVMetadataObject *transformedCode =
            [self.previewLayer
            transformedMetadataObjectForMetadataObject:code];
        [transformedCodes addObject:transformedCode];
    }
    return transformedCodes;
}

@end

```

- 1.** Create a local NSArray to store the transformed metadata objects based on the AVMetadataMachineReadableCodeObject instances passed to the delegate method.
- 2.** Create a new method to perform the translation from the device coordinate-space metadata objects into view coordinate-space objects. This method will iterate through the collection of metadata objects passed to the method, and for each you'll call the preview layer's transformedMetadataObjectForMetadataObject: method and add it to the transformedCodes array.

You now have a collection of AVMetadataMachineReadableCodeObject instances that have meaningful coordinates for constructing the user interface. Let's continue implementing this method as shown in [Listing 7.20](#).

Listing 7.20 Building the Layers

[Click here to view code image](#)

```

@implementation THPreviewView

...
- (void)didDetectCodes:(NSArray *)codes {
    NSArray *transformedCodes = [self
        transformedCodesFromCodes:codes];
    NSMutableArray *lostCodes = [self.codeLayers.allKeys

```

```

mutableCopy];           // 1

    for (AVMetadataMachineReadableCodeObject *code in
transformedCodes) {

        NSString *stringValue =
code.stringValue;                      // 2
        if (stringValue) {
            [lostCodes removeObject:stringValue];
        } else {
            continue;
        }

        NSArray *layers =
self.codeLayers[stringValue];           // 3

        if (!layers) {
            // no layers for stringValue, create new code layers
            layers = @[[self makeBoundsLayer], [self
makeCornersLayer]];

            self.codeLayers[stringValue] = layers;
            [self.previewLayer addSublayer:layers[0]];
            [self.previewLayer addSublayer:layers[1]];
        }

        CAShapeLayer *boundsLayer =
layers[0];                           // 4
        boundsLayer.path = [self
bezierPathForBounds:code.bounds].CGPath;

        NSLog(@"%@", stringValue);          // 5
    }

    for (NSString *stringValue in lostCodes)
{                                     // 6
        for (CALayer *layer in self.codeLayers[stringValue]) {
            [layer removeFromSuperlayer];
        }
        [self.codeLayers removeObjectForKey:stringValue];
    }
}

- (UIBezierPath *)bezierPathForBounds:(CGRect)bounds {
    return [UIBezierPath bezierPathWithRect:bounds];
}

- (CAShapeLayer *)makeBoundsLayer {
    CAShapeLayer *shapeLayer = [CAShapeLayer layer];

```

```

        shapeLayer.strokeColor =
            [UIColor colorWithRed:0.95f green:0.75f blue:0.06f
alpha:1.0f].CGColor;
        shapeLayer.fillColor = nil;
        shapeLayer.lineWidth = 4.0f;
        return shapeLayer;
    }

- (CASHapeLayer *)makeCornersLayer {
    CASHapeLayer *cornersLayer = [CASHapeLayer layer];
    cornersLayer.lineWidth = 2.0f;
    cornersLayer.strokeColor =
        [UIColor colorWithRed:0.172 green:0.671 blue:0.428
alpha:1.000].CGColor;
    cornersLayer.fillColor =
        [UIColor colorWithRed:0.190 green:0.753 blue:0.489
alpha:0.500].CGColor;

    return cornersLayer;
}

@end

```

- 1.** Create a mutable copy of the keys from the `codeLayers` dictionary. This array will be used to determine which layers should be removed at the end of this method.
- 2.** Look up the `stringValue` from the code. If a valid string object was returned, remove it from the `lostCodes` array. If the `stringValue` is `nil`, you simply issue a `continue` statement to continue to the next iteration of the loop because you should skip over any codes that don't have a legitimate value.
- 3.** Look up the existing array of layers for the current `stringValue`. If no entry for this value exists, create two new `CASHapeLayer` objects. A `CASHapeLayer` is a specialized `CALayer` subclass used to draw a Bezier path. The first will be used to draw the `bounds` rectangle and the other will draw the `corners` path that you'll be constructing shortly. Add an entry for the layers in the dictionary and add each layer to the `previewLayer`.
- 4.** For the `bounds` layer, create a `UIBezierPath` correlating to the object's `bounds`. Core Animation works only with Quartz types, so you ask for `UIBezierPath` for its `CGPath` property, which will

return a CGPathRef to set as the layer's path property.

5. For the purposes of this app, you log the stringValue to the console. In a real application, you'll likely want to display this value in a meaningful way to the user.
6. Finally, iterate over the remaining lostCodes entries, and for each you remove the layers from the previewLayer and remove the array entry from the dictionary.

You can now run the application. You can use the barcodes in [Table 7.2](#) to test this feature. For each detected code, you'll see its bounding rectangle drawn onscreen. If the camera's view is perpendicular to the barcode, the bounding rectangle lines up well. However, this rectangle doesn't align well when scanning at an angle. This is where the corners property comes in. In most cases the corners property provides a better source for drawing the code's geometry because it provides a dictionary containing the code's corner points. Using these points makes it much easier to construct a Bezier path that tightly aligns with the corners of the code. Let's see how you can make use of the corners property in [Listing 7.21](#).

Listing 7.21 Using the corners Property

[Click here to view code image](#)

```
@implementation THPreviewView

...
- (void)didDetectCodes:(NSArray *)codes {
    NSArray *transformedCodes = [self
        transformedCodesFromCodes:codes];
    NSMutableArray *lostCodes = [self.codeLayers.allKeys
        mutableCopy];
    for (AVMetadataMachineReadableCodeObject *code in
        transformedCodes) {
        NSString *stringValue = code.stringValue;
        if (stringValue) {
            [lostCodes removeObject:stringValue];
        } else {
            continue;
        }
    }
}
```

```

NSArray *layers = self.codeLayers[stringValue];

if (!layers) {
    // no layers for stringValue, create new code layers
    layers = @[[self makeBoundsLayer], [self
makeCornersLayer]];

    self.codeLayers[stringValue] = layers;
    [self.previewLayer addSublayer:layers[0]];
    [self.previewLayer addSublayer:layers[1]];
}

CAShapeLayer *boundsLayer = layers[0];
boundsLayer.path = [self
bezierPathForBounds:code.bounds].CGPath;

CAShapeLayer *cornersLayer =
layers[1];                                // 1
cornersLayer.path = [self
bezierPathForCorners:code.corners].CGPath;

NSLog(@"%@", "String: %@", stringValue);
}

for (NSString *stringValue in lostCodes) {
    for (CALayer *layer in self.codeLayers[stringValue]) {
        [layer removeFromSuperlayer];
    }
    [self.codeLayers removeObjectForKey:stringValue];
}
}

- (UIBezierPath *)bezierPathForCorners:(NSArray *)corners {
    UIBezierPath *path = [UIBezierPath bezierPath];
    for (int i = 0; i < corners.count; i++) {
        CGPoint point = [self
pointForCorner:corners[i]];                                // 2
        if (i == 0)                                         // 4
{
            [path moveToPoint:point];
        } else {
            [path addLineToPoint:point];
        }
    }
    [path
closePath];
5
    return path;
}

```

```

- (CGPoint)pointForCorner:(NSDictionary *)corner
{
    CGPoint point;
    CGPointMakeWithDictionaryRepresentation((CFDictionaryRef)corner,
    &point);
    return point;
}

...
@end

```

- 1.** For the `cornersLayer`, construct a `CGPath` based on the metadata object's `corners` property.
- 2.** In the `bezierPathForCorners:` method, begin by creating an empty `UIBezierPath`. Iterate over the entries in the `corners` array for each entry you construct a `CGPoint`.
- 3.** The dictionary containing the corner points has an entry for the `X` value and an entry for the `Y` value. You could pull these values out yourself and manually construct a `CGPoint`. However, the Quartz framework provides a convenient function called `CGPointMakeWithDictionaryRepresentation` to do this for you. Simply pass it the dictionary and a pointer to the `CGPoint` struct to be populated.
- 4.** Construct a Bezier path based on these points using the methods on `UIBezierPath`. For the first point you issue a `moveToPoint:` call to begin the path, and for subsequent points you call `addLineToPoint:` to connect the remaining points.
- 5.** Finally, close the path and return the `UIBezierPath` instance to the caller.

Run the application again. Now as you move your device around, you'll see that the layer tightly aligns with the corners of the barcode.

Having the capability of detecting machine-readable codes is a very welcome addition to the framework, and I expect this feature will be widely implemented in a variety of applications. I recommend experimenting with some of the other supported symbologies to get a better sense for the

capabilities of this feature. I would also recommend reading Apple's Tech Note TN2325 because it provides some additional insight and tips that will be useful to anyone implementing this functionality.

Using High Frame Rate Capture

One of the most interesting features introduced with iOS 7 was the capability of capturing high frame rate video on the latest generation of iOS devices. Capturing video at a high number of frames per second (FPS) provides a couple of interesting benefits. The first is that high FPS video often provides a greater sense of realism and clarity because movement is more accurately captured due to the increased temporal sampling rate. The enhanced detail and fluidity of motion this provides is quite striking, especially when recording fast-moving content such as a sporting event. The other, and probably more widely used, benefit is to enable high-quality slow-motion video effects. The latest generation of iOS devices supports capturing video at 60FPS (or even 120FPS in the case of the iPhone 5s), which means you can cut the playback rate to half speed while retaining a 30FPS frame rate providing a smooth playback experience. Nothing adds dramatic flair more than slow motion, and now that capability is well within your reach.

High frame rate capture is something that developers have wanted for some time. Fortunately, Apple didn't introduce it as an isolated feature, but instead provides strong support for it throughout the framework.

- **Capture:** The framework supports capturing 720p video at 60FPS, or up to 120FPS on the iPhone 5s, with video stabilization. It additionally supports an h.264 feature that enables droppable P-frames, which will ensure that your high frame-rate content plays smoothly on older devices.
- **Playback:** AVPlayer already provided support for playing content at variable playback rates; however, a significant audio processing enhancement was made to AVPlayerItem, providing control over how the audio content is processed when playing back at reduced playback rates. AVPlayerItem now has an `audioTimePitchAlgorithm` property enabling you to set the algorithm used when slowing down or speeding up the playback rate. See the API documentation for AVPlayerItem for its available options.

- **Editing:** The framework's editing features provide full support for performing scaled edits within mutable compositions. We'll discuss how to compose and edit media starting in [Chapter 8](#).
- **Export:** AV Foundation provides the capability of preserving the original frame rate, so the high FPS content will be exported, or it can perform a frame rate conversion so that all content is flattened out into standard 30FPS output.

High Frame Rate Capture Overview

In the previous chapter, you saw how to set the capture session's quality of service by specifying a session preset. You may be expecting to find some new presets supporting high frame rate capture. Unfortunately, enabling this feature isn't quite so simple. It isn't practical to extend this preset mechanism, because it would cause an explosion of presets required to account for all combinations of frame rates and size. Instead, this feature is enabled through a *parallel* configuration mechanism that was introduced in iOS 7. This new approach doesn't replace the preset model, but provides you with alternate ways of configuring the session when more fine-grained control is needed.

Earlier in this chapter you made use of `AVCaptureDeviceFormat` to determine the maximum zoom factor supported by the active capture device format. In the video zooming example, you were working with the device's `activeFormat`, which is automatically set based on the selected session preset. In addition to its *active* format, you can also ask a device for all of its *supported* formats by querying its `formats` property. An instance of `AVCaptureDeviceFormat` has a

`videoSupportedFrameRateRanges` property, which contains an array of `AVFrameRateRange` objects detailing the minimum and maximum frame rates and durations supported by the format. The basic recipe to enable high frame rate capture is to find the device's highest quality format, find its associated frame durations, and then manually set the format and frame durations values on the capture device. This will become more clear when you get into some sample code. Open the **SlowKamera_Starter** project in the [Chapter 7](#) directory, and let's begin implementing this functionality.

Enabling High Frame Rate Capture

Enabling this support is currently a bit cumbersome, but one way you could simplify this a bit is to wrap this functionality up into a category on `AVCaptureDevice` (see [Listing 7.22](#)). Let's begin by building this category.

Listing 7.22 High Frame Rate Capture Category

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>

@interface AVCaptureDevice (THAdditions)

- (BOOL)supportsHighFrameRateCapture;
- (BOOL)enableHighFrameRateCapture:(NSError **_)error;

@end
```

You'll create a category on `AVCaptureDevice` by adding two methods: one to determine if high frame rate capture is supported by the current device and another to enable the feature if supported. Let's move on to the implementation.

Before providing the implementation of the category methods themselves, you'll start by creating a private class inside the `AVCaptureDevice+THAdditions.m` file called `THQualityOfService` that will be used to simplify the category method implementations (see [Listing 7.23](#)).

Listing 7.23 Private THQualityOfService Class

[Click here to view code image](#)

```
#import "AVCaptureDevice+THAdditions.h"
#import "TSError.h"

@interface THQualityOfService : NSObject

@property(strong, nonatomic, readonly) AVCaptureDeviceFormat
*format;
@property(strong, nonatomic, readonly) AVFrameRateRange
*frameRateRange;
@property(nonatomic, readonly) BOOL isHighFrameRate;

+ (instancetype)qosWithFormat:(AVCaptureDeviceFormat *)format
```

```

        frameRateRange: (AVFrameRateRange *) frameRateRange;

- (BOOL)isHighFrameRate;

@end

@implementation THQualityOfService

+ (instancetype)qosWithFormat:(AVCaptureDeviceFormat *)format
    frameRateRange:(AVFrameRateRange *)frameRateRange {

    return [[self alloc] initWithFormat:format
frameRateRange:frameRateRange];
}

- (instancetype)initWithFormat:(AVCaptureDeviceFormat *)format
    frameRateRange:(AVFrameRateRange *)frameRateRange
{
    self = [super init];
    if (self) {
        _format = format;
        _frameRateRange = frameRateRange;
    }
    return self;
}

- (BOOL)isHighFrameRate {
    return self.frameRateRange.maxFrameRate > 30.0f;
}

@end

```

To enable high frame rate capture, you'll need to look through a capture device's available formats and find the highest supported AVCaptureDeviceFormat and its highest AVFrameRateRange. The THQualityOfService is used to store these values and simplify the implementation of the public category methods. Let's look at the implementation of the first category method in [Listing 7.24](#).

Listing 7.24 Determining High FPS Support

[Click here to view code image](#)

```

@implementation AVCaptureDevice (THAdditions)

- (BOOL)supportsHighFrameRateCapture {
    if (![self hasMediaType:AVMediaTypeVideo])

```

```

{
    return NO; // 1
}
return [self
findHighestQualityOfService].isHighFrameRate; // 2
}

- (THQualityOfService *)findHighestQualityOfService {

    AVCaptureDeviceFormat *maxFormat = nil;
    AVFrameRateRange *maxFrameRateRange = nil;

    for (AVCaptureDeviceFormat *format in self.formats) {

        FourCharCode codecType // 3
        =
        CMVideoFormatDescriptionGetCodecType(format.formatDesc);

        if (codecType ==
kCVPixelFormatType_420YpCbCr8BiPlanarVideoRange) {

            NSArray *frameRateRanges =
format.videoSupportedFrameRateRanges;

            for (AVFrameRateRange *range in frameRateRanges)
{ // 4
                if (range.maxFrameRate >
maxFrameRateRange.maxFrameRate) {
                    maxFormat = format;
                    maxFrameRateRange = range;
                }
            }
        }
    }

    return [THQualityOfService
qosWithFormat:maxFormat // 6
frameRateRange:maxFrameRateRange];
}

...
@end

```

1. Determine if the `AVCaptureDevice` instance is a video device by asking if it has the media type of `AVMediaTypeVideo`. If not, return `NO` from this method.
2. Call the `findHighestQualityOfService` method to determine

the maximum format and frame rates this camera device supports. You ask the `THQualityOfService` object if it supports high frame rate capture and return the value.

3. Iterate over all the capture device's supported formats and for each get the `codecType` from its `formatDescription`. A `CMFormatDescriptionRef` is an opaque type from the Core Media that provides a variety of interesting details about the format object. You'll want to process only the formats that have a `codeType` of `420YpCbCr8BiPlanarVideoRange`, which will limit the search to just the video formats.
4. Iterate over the format's collection of `AVFrameRateRange` objects returned from its `videoSupportedFrameRateRanges` property. For each, you determine if its `maxFrameRate` is greater than the current maximum. The ultimate goal is to find the highest format and frame rate support provided by this camera device.
5. Finally, return a new instance of the internal `THQualityOfService` capturing the highest format and frame rate range supported.

You now have the means to determine the highest level of support provided by the active camera, so let's look at how to actually enable this feature in [Listing 7.25](#).

Listing 7.25 Enabling High Frame Rate Capture

[Click here to view code image](#)

```
- (BOOL)enableMaxFrameRateCapture:(NSError **)error {  
  
    THQualityOfService *qos = [self findHighestQualityOfService];  
  
    if (!qos.isHighFrameRate) // 1  
    {  
        if (error) {  
            NSString *message = @"Device does not support high FPS  
capture";  
            NSDictionary *userInfo = @{@"NSLocalizedDescriptionKey" :  
message};  
  
            NSUInteger code =  
THCameraErrorHighFrameRateCaptureNotSupported;
```

```

        *error = [NSError errorWithDomain:THCameraErrorDomain
                                      code:code
                                     userInfo:userInfo];
    }
    return NO;
}

if ([self lockForConfiguration:error])
{
    // 2

    CMTIME minFrameDuration =
qos.frameRateRange.minFrameDuration;

    self.activeFormat =
qos.format;                                // 3
    self.activeVideoMinFrameDuration =
minFrameDuration;                         // 4
    self.activeVideoMaxFrameDuration = minFrameDuration;

    [self unlockForConfiguration];
    return YES;
}
return NO;
}

```

- 1.** Begin by finding the maximum quality of service provided by the device. If it does not support high frame rate capture, you populate the error pointer and return NO.
- 2.** Prior to modifying the capture device, you need to lock the device for configuration.
- 3.** Set the devices activeFormat to the retrieved AVCaptureDeviceFormat.
- 4.** Pin both the minimum and maximum frame durations to that of the values defined by the AVFrameRateRange. AV Foundation generally deals in frame durations, specified as instances of a CMTIME, and not frame rates. The minFrameDuration is the reciprocal of the maxFrameRate. For instance, a frame rate of 60FPS would be expressed in duration as 1/60th of a second.

With the category complete, let's put this to work in the application's THCameraController class (see [Listing 7.26](#)).

Listing 7.26 Using AVCaptureDevice+THAdditions

[Click here to view code image](#)

```
#import "THCameraController.h"
#import "AVCaptureDevice+THAdditions.h"

@implementation THCameraController

- (BOOL)cameraSupportsHighFrameRateCapture {
    return [self.activeCamera supportsHighFrameRateCapture];
}

- (BOOL)enableHighFrameRateCapture {
    NSError *error;
    BOOL enabled = [self.activeCamera
        enableMaxFrameRateCapture:&error];
    if (!enabled) {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
    return enabled;
}

@end
```

If you have a latest generation device, you can see this feature in action. Launch the application and record several seconds of video. When you hit the Stop button, the video will be presented in a player with a rate control. Experiment with the different rate options to see how great high frame rate video looks at a variety of playback rates.

High frame rate capture had been a widely requested feature, and starting with iOS 7 this capability is now at your disposal. In addition to capture, AV Foundation provides full support for high frame rate content throughout all aspects of the framework, which provides you with the creative tools needed to build an exciting new class of capture apps.

Processing Video

In [Chapter 6](#), you used the [AVCaptureMovieFileOutput class to capture a QuickTime movie](#). This class provides an easy way of capturing video data from the built-in cameras, but it doesn't provide the capability of interacting with the video data itself, which is needed in many scenarios. Consider the needs of an app like Apple's Photo Booth that applies real-time

video effects to the video stream or an augmented reality app like Sphero's Sharky the Beaver, which projects an interactive character into any environment captured by the camera. Both of these examples require more low-level control over the captured video data than what is provided by `AVCaptureMovieFileOutput`. When you need this level of control, you'll turn to the lowest-level video capture output provided by the framework called `AVCaptureVideoDataOutput`.

`AVCaptureVideoDataOutput` is an `AVCaptureOutput` subclass providing direct access to the video frames as they are captured by the camera sensor. This is an incredibly powerful capability because you have full control over the video data's format, timing, and metadata, enabling you to manipulate the video content as needed. Most commonly, this processing is performed using OpenGL ES or Core Image, but even Quartz may suffice for some simple processing. In the sample app you'll be building shortly, I'll show how to integrate with OpenGL ES because this is widely used for processing video data, but in the next chapter we'll also look at how to integrate with Core Image and leverage its powerful collection of filters.

Note

AV Foundation provides a low-level capture output for working with audio data as well as called `AVCaptureAudioDataOutput`. We'll focus solely on `AVCaptureVideoDataOutput` in this chapter, but you'll make use of its audio sibling in the next chapter when we discuss using `AVAssetReader` and `AVAssetWriter`.

Using an `AVCaptureVideoDataOutput` is similar to `AVCaptureMetadataOutput`, which you used earlier in the chapter, with the most notable difference being its delegate callback. Instead of outputting instances of `AVMetadataObject`, an `AVCaptureVideoDataOutput` outputs objects containing the video data via its `AVCaptureVideoDataOutputSampleBufferDelegate` protocol.

`AVCaptureVideoDataOutputSampleBufferDelegate` defines the following methods:

- **`captureOutput:didOutputSampleBuffer:fromConnection:`**

This method is called whenever a new video frame is written. The data will be decoded or reencoded based on the configuration of the video data output's `videoSettings` property.

- **`captureOutput:didDropSampleBuffer:fromConnection`**

This method is called whenever a late video frame was dropped. The most common reason this is called is due to taking too much processing time in the `didOutputSampleBuffer:` call. You must perform your processing as efficiently as possible, or you will eventually fail to receive any buffers.

The most important argument in both methods is the one relating to the sample buffer. The sample buffer is provided in the form of an object called `CMSampleBuffer`, which was briefly discussed in [Chapter 6](#) when we worked with `AVCaptureStillImageOutput`. Having a solid understanding of this type is essential, so let's look at the details of what this object is and what it provides.

Understanding CMSampleBuffer

A `CMSampleBuffer` is a Core Foundation-style object provided by the Core Media framework that is used to shuttle digital samples through the media pipeline. `CMSampleBuffer` acts as a wrapper over the underlying sample data and provides format and timing information, along with any additional metadata required to interpret and process the data. Let's begin by looking at the sample data provided by a `CMSampleBuffer`.

Sample Data

When working with `AVCaptureVideoDataOutput`, the sample buffer will contain a `CVPixelBuffer`, which is a Core Video object providing the raw pixel data for a single video frame. The following example illustrates how you could directly manipulate the contents of a `CVPixelBuffer` to apply a grayscale effect to the captured image buffer.

[Click here to view code image](#)

```
const int BYTES_PER_PIXEL = 4;

CMSampleBufferRef sampleBuffer = // obtained sample buffer

CVPixelBufferRef pixelBuffer
```

```

=                                     // 1
    CMSampleBufferGetImageBuffer(sampleBuffer);

    CVPixelBufferLockBaseAddress( pixelBuffer,
        0);                           // 2

    size_t bufferWidth =
    CVPixelBufferGetWidth(pixelBuffer);           // 3
    size_t bufferHeight = CVPixelBufferGetHeight(pixelBuffer);

    unsigned char *pixel
    =                               // 4
        (unsigned char *)CVPixelBufferGetBaseAddress(pixelBuffer);
    unsigned char grayPixel;

    for (int row = 0; row < bufferHeight; row++)
    {                         // 5
        for(int column = 0; column < bufferWidth; column++) {
            grayPixel = (pixel[0] + pixel[1] + pixel[2]) / 3;
            pixel[0] = pixel[1] = pixel[2] = grayPixel;
            pixel += BYTES_PER_PIXEL;
        }
    }

    CVPixelBufferUnlockBaseAddress(pixelBuffer,
        0);                           // 6

    // Process grayscale video frame

```

There is quite bit of new information presented in this example, so let's walk through some of the details.

- 1.** You begin by getting the underlying `CVPixelBuffer` from the `CMSampleBufferRef` by using the `CMSampleBufferGetImageBuffer` function. The `CVPixelBuffer` holds the pixel data in main memory, providing the opportunity to manipulate its contents.
- 2.** Prior to interacting with the `CVPixelBuffer` data, you must call `CVPixelBufferLockBaseAddress` to obtain a lock on that block of memory.
- 3.** Determine the width and height of the pixel buffer using the `CVPixelBufferGetWidth` and `CVPixelBufferGetHeight` functions so you'll be able to iterate over its rows and columns.
- 4.** Get the base address pointer for this pixel buffer using the `CVPixelBufferGetBaseAddress` function. This enables you to

index into this buffer and iterate over its data.

5. Iterate over the rows and columns of pixels in the buffer and perform some simple grayscale averaging of the RGB pixels.
6. Finally, you need to release the lock you obtained in step 2, by calling the `CVPixelBufferUnlockBaseAddress` function.

From here, you could convert this buffer into a `CGImageRef` or `UIImage` or perform any additional image processing needed. I would recommend looking at the additional `CVPixelBuffer` functions available in `CVPixelBuffer.h`.

Format Descriptions

In addition to the raw media samples themselves, `CMSampleBuffer` also provides access to format information about the samples in the form of an object called `CMFormatDescription`. A variety of functions are defined in `CMFormatDescription.h` that enable you to access details about the media samples. The functions contained within this header that are prefixed with `CMFormatDescription` apply generally to all media types, and you'll also find functions prefixed with `CMVideoFormatDescription` and `CMAudioFormatDescription` for getting the details for video and audio, respectively.

Let's look at one way of using a `CMFormatDescription` to differentiate between audio and video data.

[Click here to view code image](#)

```
CMFormatDescriptionRef formatDescription =
    CMSampleBufferGetFormatDescription(sampleBuffer);

CMMediaType mediaType =
    CMFormatDescriptionGetMediaType(formatDescription);

if (mediaType == kCMMediaType_Video) {
    CVPixelBufferRef pixelBuffer =
        CMSampleBufferGetImageBuffer(sampleBuffer);
    // Process the frame of video
} else if (mediaType == kCMMediaType_Audio) {
    CMBlockBufferRef blockBuffer =
        CMSampleBufferGetDataBuffer(sampleBuffer);
    // Process audio samples
}
```

Timing

A CMSampleBuffer also provides timing information about the media sample. The timing information can be extracted using the CMSampleBufferGetPresentationTimeStamp and CMSampleBufferGetDecodeTimeStamp functions to get the original presentation timestamp and the decode timestamp, respectively. We'll look more at using these functions when discussing AVAssetReader and AVAssetWriter in the next chapter.

Metadata Attachments

Core Media also provides a metadata protocol in the form of a CMAttachment defined in CMAttachment.h. This API provides the facilities needed to read and write lower-level metadata, such as exchangeable image file format (Exif) tags. For instance, the following example shows you can retrieve the Exif metadata from a given CMSampleBuffer.

[Click here to view code image](#)

```
CFDictionaryRef exifAttachments =
    (CFDictionaryRef)CMGetAttachment(sampleBuffer,
                                    kCGImagePropertyExifDictionary,
                                    NULL);
```

If you printed this dictionary to the console, you would see output similar to the following example.

[Click here to view code image](#)

```
{
    ApertureValue = "2.526068811667587";
    BrightnessValue = "-0.4554591284958377";
    ExposureMode = 0;
    ExposureProgram = 2;
    ExposureTime = "0.04166666666666666";
    FNumber = "2.4";
    Flash = 32;
    FocalLenIn35mmFilm = 35;
    FocalLength = "2.18";
    ISOSpeedRatings =      (
        800
    );
    LensMake = Apple;
    LensModel = "iPhone 5 front camera 2.18mm f/2.4";
```

```

    LensSpecification =      (
        "2.18",
        "2.18",
        "2.4",
        "2.4"
    );
    MeteringMode = 5;
    PixelXDimension = 640;
    PixelYDimension = 480;
    SceneType = 1;
    SensingMethod = 2;
    ShutterSpeedValue = "4.584985584026477";
    WhiteBalance = 0;
}

```

`CMSampleBuffer` and its related types play an important role when working with advanced AV Foundation use cases. We'll put this into action shortly, but we will see some additional ways of working with this type in the next chapter.

Using `AVCaptureVideoDataOutput`

Let's look at a sample project and see how you can put `AVCaptureVideoDataOutput` to use. You'll find a sample project in the [Chapter 7](#) directory called **CubeKamera_Starter**. This project demonstrates some techniques used to integrate AV Foundation and OpenGL ES. The CubeKamera app captures video from the front-facing camera and maps those video frames as OpenGL textures onto a spinning cube.

Let's begin by looking at the interface for the project's `THCameraController` class in [Listing 7.27](#).

Listing 7.27 `THCameraController` Interface

[Click here to view code image](#)

```

#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THTextureDelegate <NSObject>
- (void)textureCreatedWithTarget:(GLenum)target name:(GLuint)name;
@end

@interface THCameraController : THBaseCameraController
- (instancetype)initWithContext:(EAGLContext *)context;

```

```
@property (weak, nonatomic) id <THTextureDelegate>
textureDelegate;

@end
```

The interface for this class looks similar to the ones you've seen throughout this chapter, with the exception of some of the OpenGL ES types defined. The `THCameraController` is created with an instance of `EAGLContext`. This object provides the *rendering* context that manages the state and resources needed to draw using OpenGL ES. The header also defines a `THTextureDelegate` protocol that will be called whenever a new texture, an immutable OpenGL ES image, is created. Let's move on to the implementation in [Listing 7.28](#).

Listing 7.28 Configuring the Capture Output

[Click here to view code image](#)

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController : NSObject
<AVCaptureVideoDataOutputSampleBufferDelegate>

@property (weak, nonatomic) EAGLContext *context;
@property (strong, nonatomic) AVCaptureVideoDataOutput
*videoDataOutput;

@end

@implementation THCameraController

- (instancetype)initWithContext:(EAGLContext *)context {
    self = [super init];
    if (self) {
        _context = context;
    }
    return self;
}

- (NSString *)sessionPreset // 1
{
    return AVCaptureSessionPreset640x480;
}

- (BOOL)setupSessionOutputs:(NSError **)error {
```

```

        self.videoDataOutput = [[AVCaptureVideoDataOutput alloc]
init];

        self.videoDataOutput.videoSettings
= // 2
@{ (id)kCVPixelBufferPixelFormatTypeKey :
@(kCVPixelFormatType_32BGRA)};

        [self.videoDataOutput
setSampleBufferDelegate:self // 3
queue:dispatch_get_main_queue];

        if ([self.captureSession canAddOutput:self.videoDataOutput])
{
    // 4
    [self.captureSession addOutput:self.videoDataOutput];
    return YES;
}

        return NO;
}

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
fromConnection:(AVCaptureConnection *)connection {
}

@end

```

- 1.** This application doesn't require high-resolution video, so you override the `sessionPreset` method to return `AVCaptureSessionPreset640x480`.
- 2.** Create a new instance of `AVCaptureVideoDataOutput` and customize its `videoSettings` dictionary. The camera's native format is bi-planar 420v. If you'll recall the discussion in [Chapter 1](#), "[Getting Started with AV Foundation](#)," on chroma subsampling, this format separates the luminance and chrominance and subsamples the color in both the horizontal and vertical directions. Although it is certainly possible to work directly in this native format, it is often preferable to use BGRA when working with OpenGL ES. Be aware this format conversion does incur a slight performance penalty.
- 3.** Set the output's delegate, which is `self` because this class adopts the

`AVCaptureVideoDataOutputSampleBufferDelegate` protocol, and also specify the dispatch queue on which the delegate's methods should be called. The dispatch queue specified for the `queue` argument must be a *serial* queue. In this case, I'm using the *main queue* because it works best for this application, but in many cases, this will be a dedicated video-processing queue.

4. Finally, perform the standard test to determine if the output can be added to the session, and if so, you add it and return YES. If the test failed, you return NO.

So far, this looks similar to what you've done in the previous examples. Let's move on and talk about some of the OpenGL ES integration points. Coverage of OpenGL ES is well outside the scope of this book, but as an AV Foundation developer, you'll find there are many advanced cases where OpenGL ES is the only viable solution to provide the control and capabilities you need for high-performance video applications. As such, we'll cover the facilities provided by AV Foundation's underlying frameworks that act as a bridge into OpenGL ES. For a complete overview of OpenGL ES 2.0, I recommend reading *Learning OpenGL ES for iOS* by Erik Buck (2012, Boston: Addison-Wesley).

Core Video provides an object type called `CVOpenGLTextureCache` that acts as a bridge between Core Video pixel buffers and OpenGL ES textures. The purpose of the cache is to eliminate the costly data transfers from the CPU to the GPU (and potentially back again) that you would need to do without it. Let's begin by creating the texture cache as shown in [Listing 7.29](#).

Listing 7.29 Creating the `OpenGLTextureCache`

[Click here to view code image](#)

```
@interface THCameraController ()  
<AVCaptureVideoDataOutputSampleBufferDelegate>  
  
@property (weak, nonatomic) EAGLContext *context;  
@property (strong, nonatomic) AVCaptureVideoDataOutput  
*videoDataOutput;  
  
@property (nonatomic) CVOpenGLTextureCacheRef  
textureCache; // 1  
@property (nonatomic) CVOpenGLTextureRef cameraTexture;
```

```

@end

@implementation THCameraController

- (instancetype)initWithContext:(EAGLContext *)context {
    self = [super init];
    if (self) {
        _context = context;
        CVReturn err =
CVOpenGLESTextureCacheCreate(kCFAllocatorDefault,           // 2
                                NULL,
                                _context,
                                NULL,
&_textureCache);
        if (err != kCVReturnSuccess)
    {                               // 3
        NSLog(@"Error creating texture cache. %d", err);
    }
}
return self;

}

```

- 1.** Add two new properties. The first is for the `CVOpenGLESTextureCacheRef` itself and the second is for the `CVOpenGLESTextureRef` object that will be created in the `AVCaptureVideoDataOutput` delegate callback.
- 2.** Use the `CVOpenGLESTextureCacheCreate` function to create a new instance of the cache. The key arguments provided to this function are the backing `EAGLContext` and the `textureCache` pointer.
- 3.** It's a good idea to check the return value from this function. In this case, if it returns anything other than `kCVReturnSuccess`, you just log the error to the console. I would suggest handling this more robustly in a production app.

With the texture cache created, you're able to move on and actually begin creating some textures. The function to create a texture from a `CVPixelBuffer` is called

`CVOpenGLESTextureCacheCreateTextureFromImage`. This function will be used inside the delegate callback, creating a new texture for each video frame. [Listing 7.30](#) illustrates how this is used.

Listing 7.30 Creating OpenGL ES Textures

[Click here to view code image](#)

```
- (void)captureOutput: (AVCaptureOutput *)captureOutput
didOutputSampleBuffer: (CMSampleBufferRef)sampleBuffer
fromConnection: (AVCaptureConnection *)connection {

    CVReturn err;
    CVImageBufferRef pixelBuffer
    = CMSampleBufferGetImageBuffer(sampleBuffer); // 1

    CMFormatDescriptionRef formatDescription
    = CMSampleBufferGetFormatDescription(sampleBuffer);
    CMVideoDimensions dimensions =
        CMVideoFormatDescriptionGetDimensions(formatDescription);

    err =
    CVOpenGLESTextureCacheCreateTextureFromImage(kCFAllocatorDefault,
// 3
                                         _textureCache,
                                         pixelBuffer,
                                         NULL,
                                         GL_TEXTURE_2D,
                                         GL_RGBA,
                                         dimensions.height,
                                         dimensions.height,
                                         GL_BGRA,
                                         GL_UNSIGNED_BYTE,
                                         0,
                                         &_cameraTexture);

    if (!err) {
        GLenum target =
        CVOpenGLESTextureGetTarget(_cameraTexture); // 4
        GLuint name = CVOpenGLESTextureGetName(_cameraTexture);
        [self.textureDelegate textureCreatedWithTarget:target
name:name];
// 5
    } else {
        NSLog(@"Error at
CVOpenGLESTextureCacheCreateTextureFromImage %d", err);
    }

    [self cleanupTextures];
}

- (void)cleanupTextures
{ // 6
```

```

    if (_cameraTexture) {
        CFRelease(_cameraTexture);
        _cameraTexture = NULL;
    }
    CVOpenGLTextureCacheFlush(_textureCache, 0);

}

```

- 1.** Begin by getting the underlying `CVImageBuffer` from the captured `CMSampleBuffer`. (`CVImageBuffer` is a `typedef` for a `CVPixelBufferRef`.)
- 2.** Get the `CMFormatDescription` from the `CMSampleBuffer`. You use the format description to get the video frame's dimensions using the `CMVideoFormatDescriptionGetDimensions` function. This returns a `CMVideoDimensions` struct containing the width and height.
- 3.** Create the OpenGL ES texture from the `CVPixelBuffer` using the `CVOpenGLTextureCacheCreateTextureFromImage` function. You'll notice that you're passing `dimensions.height` for both the width and height arguments. This isn't a typo, but rather a bit of a cheat, because you want to clip the video horizontally so it's a perfect square. There are alternative ways to perform this cropping, but this is a simple and suitable approach for this example.
- 4.** Get the target and name from the `CVOpenGLTextureRef`. These will be needed to properly bind the texture object to the surfaces of the spinning cube.
- 5.** Call the delegate's `textureCreatedWithTarget:name` method where the delegate will perform the actual GL texture binding.
- 6.** Finally, call the private `cleanupTextures` to release the texture and flush the texture cache.

The application is complete and ready for a trial run. Run the application and you should see a continually spinning cube with the captured video frames mapped to its face. This is a simple application, but provides you with a good understanding of how to integrate AV Foundation and OpenGL ES. You'll find many times where the two technologies work very well together.

`AVCaptureVideoDataOutput` provides the interface to access the video frames at the time they are being captured. This provides complete control

over how the data is presented or processed, enabling you to build incredibly powerful video applications. We'll come back to this topic again in the next chapter where you'll see how you can record the results of your custom video processing.

Summary

We covered many powerful features that can add a great deal of polish and excitement to your applications. Although we've looked at these features in isolation, in practice you'll find that many of these capabilities are highly complimentary and can be put to good use in a single application. The features and capabilities that we've covered in [Chapters 6](#) and [7](#) will provide you with the tools you need to build the next generation of amazing capture applications.

Challenge

Take some of the features that we covered in this chapter and integrate them into the Kamera application from [Chapter 6](#). Use the iOS Camera app as your guide to see how it makes use of zooming and face detection, and for an extra bit of polish you could enable high frame rate capture. As always, experiment and let your own creativity be your guide.

8. Reading and Writing Media

AV Foundation provides a great deal of high-level functionality for performing dedicated A/V tasks such as playback, capture, and editing. These high-level features are part of what makes AV Foundation so compelling; you can use these features to build powerful media applications without getting into the low-level details of how the media is processed. However, at times the framework's high-level features alone may not meet your needs, and you'll need to dig a little deeper. In this chapter we explore the framework's low-level reading and writing facilities and discuss some of the common scenarios in which they can be useful when you're building AV Foundation applications.

Overview

AV Foundation provides a broad set of functionality covering the use cases you'll commonly encounter when building media apps. Its dedicated high-level features will generally be your primary way of working with the framework, but as you start building more advanced media apps, you'll encounter use cases requiring functionality not natively supported by the framework. Does this mean you're out of luck and you'll need to turn elsewhere for a solution? No, it just means you'll need to drop to a lower level of functionality provided by the framework's `AVAssetReader` and `AVAssetWriter` classes (see [Figure 8.1](#)). These classes provide you with the capability of working directly with the media samples, which opens up a world of possibilities.

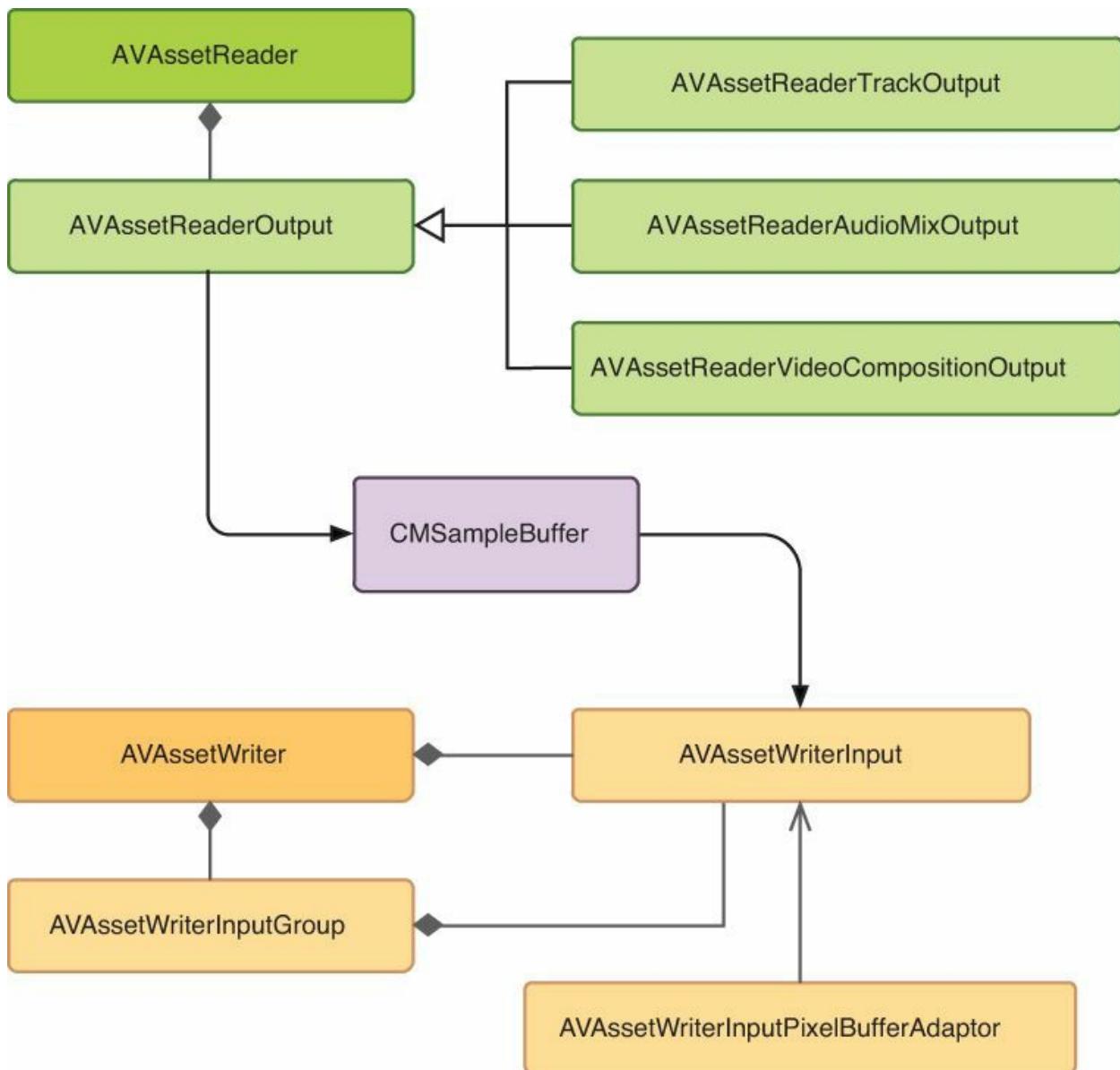


Figure 8.1 AVAssetReader and AVAssetWriter classes

AVAssetReader

AVAssetReader is used to read media samples from an instance of AVAsset. It's configured with one or more instances of AVAssetReaderOutput, which provide access to the audio samples and video frames via its `copyNextSampleBuffer` method.

AVAssetReaderOutput is an abstract class, but the framework provides three concrete instances that enable you to read the decoded media samples from a specific AVAssetTrack and also the mixed output from multiple

audio tracks or the composited output from multiple video tracks. An asset reader’s internal pipelines are multithreaded to continually fetch the next available samples, helping to minimize latency when you request them. Despite providing low-latency retrieval, it is not intended for real-time operations such as playback.

Note

An AVAssetReader can target only the media samples contained within a single asset. If you need to read samples from multiple file-based assets at the same time, you can compose them together in a subclass of AVAsset called AVComposition, which is discussed in the next chapter.

AVAssetWriter

AVAssetWriter is the counterpart to AVAssetReader and is used to encode and write media to a container file such as an MPEG-4 or a QuickTime file. It’s configured with one or more AVAssetWriterInput objects, which are used to append CMSampleBuffer objects containing the media samples to be written to the container. An AVAssetWriterInput is configured to handle a specific media type, such as audio or video, and the samples appended to it produce an individual AVAssetTrack within the final output. When working with an AVAssetWriterInput configured to process video samples, you’ll frequently use a specialized adaptor object called AVAssetWriterInputPixelBufferAdaptor. This class provides optimized performance when appending video samples that are packaged as CVPixelBuffer objects. Inputs can also be grouped into mutually exclusive arrangements using an AVAssetWriterInputGroup. This enables you to create assets containing language-specific media tracks that can be selected at playback using the AVMediaSelectionGroup and AVMediaSelectionOption classes that were discussed in [Chapter 4](#), “[Playing Video](#).”

AVAssetWriter provides automatic support for interleaving media samples. One way the samples could be written to disk is to write them sequentially, as shown in [Figure 8.2](#). All the media samples would be captured, but this would result in an inefficient arrangement of the data

because the samples that should be presented together are far apart from each other. This makes it more difficult for the storage device to efficiently read the data and would have an adverse effect on the asset's playback and seeking performance.

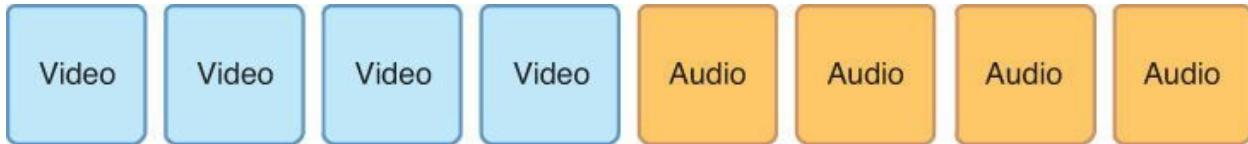


Figure 8.2 Noninterleaved sample layout

The better way to arrange these samples is in an interleaved fashion, as shown in [Figure 8.3](#). To maintain the appropriate interleaving pattern, an `AVAssetWriterInput` provides a `readyForMoreMediaData` property that indicates if the input can append more data while maintaining the needed interleaving. You can append new samples to a writer input only when this property is YES.



Figure 8.3 Interleaved sample layout

AVAssetWriter can be used for both real-time and offline operations, but you'll use a different approach in each scenario to append sample buffers to the writer's inputs:

- **Real-time:** When working with a real-time source, such as writing samples captured from an `AVCaptureVideoDataOutput`, an `AVAssetWriterInput` should have its `expectsMediaDataInRealTime` property set to YES to ensure its `readyForMoreMediaData` value is calculated appropriately. Indicating that you are writing from a real-time source optimizes the writer so that it places a greater priority on writing samples quickly in contrast to maintaining ideal interleaving. This optimization works well because the incoming data is naturally interleaved as both the audio and video samples are being captured at approximately the same rate.
- **Offline:** When reading media from an offline source, such as reading sample buffers from an `AVAssetReader`, you'll still need to observe the status of the writer input's `readyForMoreMediaData` property

before appending samples, but you'll use its `requestMediaDataWhenReadyOnQueue:usingBlock:` method to control the supply of data. The block passed to this method will be called continually as the writer input is ready to append more samples, at which time you'll retrieve and append the next samples from the source.

Reading and Writing Example

Let's work through a basic example showing how to use `AVAssetReader` and `AVAssetWriter` in an offline scenario. The example reads the samples from an asset's video track using `AVAssetReader` and writes them to a new QuickTime movie file using `AVAssetWriter`. This is admittedly a contrived example, but it illustrates the basic steps involved when using these classes together. Let's begin with the setup and configuration of the `AVAssetReader`.

[Click here to view code image](#)

```
AVAsset *asset = // Asynchronously loaded video asset
AVAssetTrack *track =
    [[asset tracksWithMediaType:AVMediaTypeVideo] firstObject];

self.assetReader =
    [[AVAssetReader alloc] initWithAsset:asset error:nil];

NSDictionary *readerOutputSettings = @{
    (id)kCVPixelBufferPixelFormatTypeKey :
    @(kCVPixelFormatType_32BGRA)
};

AVAssetReaderTrackOutput *trackOutput =
    [[AVAssetReaderTrackOutput alloc] initWithTrack:track
                                         outputSettings:readerOutput];

[self.assetReader addOutput:trackOutput];

[self.assetReader startReading];
```

The example begins by creating a new `AVAssetReader`, passing it the `AVAsset` instance to read. It creates an `AVAssetReaderTrackOutput` to read the samples from the asset's video track, decompressing the video frames into BGRA format. It adds the output to the reader and calls its `startReading` method to begin the reading process.

Next, let's create and configure the AVAssetWriter.

[Click here to view code image](#)

```
NSURL *outputURL = // Destination output URL

self.assetWriter = [[AVAssetWriter alloc] initWithURL:outputURL
                                             fileType:AVFileTypeQu:
                                             error:nil];

NSDictionary *writerOutputSettings = @{
    AVVideoCodecKey: AVVideoCodecH264,
    AVVideoWidthKey: @1280,
    AVVideoHeightKey: @720,
    AVVideoCompressionPropertiesKey: @{
        AVVideoMaxKeyFrameIntervalKey: @1,
        AVVideoAverageBitRateKey: @10500000,
        AVVideoProfileLevelKey: AVVideoProfileLevelH264Main31,
    }
};

AVAssetWriterInput *writerInput =
[[AVAssetWriterInput alloc] initWithMediaType:AVMediaTypeVideo
                                outputSettings:writerOutputSettings];

[self.assetWriter addInput:writerInput];

[self.assetWriter startWriting];
```

The example creates a new `AVAssetWriter`, passing it an output URL where the new file should be written along with the desired file type. It creates a new `AVAssetWriterInput` with the appropriate media type and output settings to create a 720p H.264 video. It adds the input to the writer and calls its `startWriting` method.

Note

A distinct advantage `AVAssetWriter` provides over `AVAssetExportSession` is that it provides fine-grained control over the compression settings you use when encoding its output. This enables you to specify settings such as the key frame interval, video bit rate, H.264 profile, pixel aspect ratio, and clean aperture.

With the `AVAssetReader` and `AVAssetWriter` objects set up, it's time to begin a new writing session to read the samples from the source asset and

write them to a new one. The example illustrates using a *pull model*, where you'll pull the samples from the source when the writer input is ready to append more samples. This is the model you'll use when writing samples from a non real-time source.

[Click here to view code image](#)

```
// Serial Queue
dispatch_queue_t dispatchQueue =
    dispatch_queue_create("com.tapharmonic.WriterQueue", NULL);

[self.assetWriter startSessionAtSourceTime:kCMTimeZero];
[writerInput requestMediaDataWhenReadyOnQueue:dispatchQueue
usingBlock:^{

    BOOL complete = NO;

    while ([writerInput isReadyForMoreMediaData] && !complete) {

        CMSampleBufferRef sampleBuffer = [trackOutput
copyNextSampleBuffer];

        if (sampleBuffer) {
            BOOL result = [writerInput
appendSampleBuffer:sampleBuffer];
            CFRelease(sampleBuffer);
            complete = !result;
        } else {
            [writerInput markAsFinished];
            complete = YES;
        }
    }

    if (complete) {
        [self.assetWriter finishWritingWithCompletionHandler:^{
            AVAssetWriterStatus status = self.assetWriter.status;
            if (status == AVAssetWriterStatusCompleted) {
                // Handle success case
            } else {
                // Handle failure case
            }
        }];
    }
}];
```

The example begins by starting a new writing session using the `startSessionAtSourceTime:` method, passing `kCMTimeZero` as the start time of the source samples. The block passed to the

`requestMediaDataWhenReadyOnQueue:usingBlock:` will be called continually as the writer input is ready to append more samples. In each invocation, while the input is ready for more data, it copies the available samples from the track output and appends them to the input. When all samples have been copied from the track output, it marks the `AVAssetWriterInput` as finished and indicates the appending is complete. Finally, it calls

`finishWritingWithCompletionHandler:` to finalize the writing session. The asset writer's `status` property can be queried in the completion handler to determine whether the writing session was completed successfully, failed, or was cancelled.

You now have a better understanding of the `AVAssetReader` and `AVAssetWriter` classes. The preceding code example provided the basic pattern you'll use when performing offline processing using these classes. Let's move on and discuss some more concrete, real-world examples so you can get a better sense for the value of `AVAssetReader` and `AVAssetWriter`.

Building an Audio Waveform View

A common requirement of many audio and video apps is to provide a graphical rendering of an audio waveform (see [Figure 8.4](#)). This makes it easier to visualize the audio tracks so you can more easily scrub to or perform edits at the desired location. In this section we'll discuss how you can build this feature using AV Foundation.

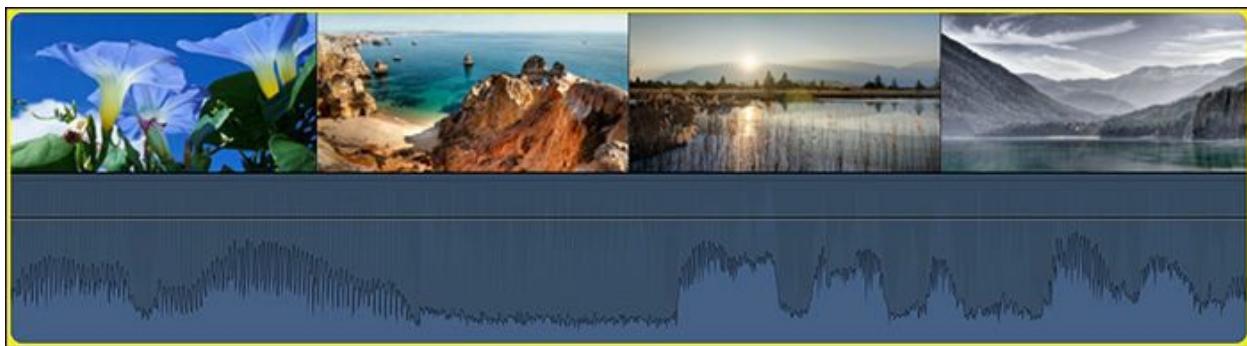


Figure 8.4 Final Cut Pro X waveform display

The basic recipe for drawing waveforms involves three main steps:

- 1. Read:** The first step is to read the audio samples to be rendered. You'll need to read, and possibly decompress, the audio data as linear PCM.

Recall from [Chapter 1](#), “[Getting Started with AV Foundation](#),” that linear PCM is the format of the uncompressed audio samples.

2. Reduce: The number of samples read will be far more than what can be rendered onscreen. Consider a mono audio file recorded at a sample rate of 44.1kHz. A single second of audio will have 44,100 samples, which is far more samples than we have pixels. A process of reduction must be applied to this sample set. This process typically involves segmenting the total number of samples into smaller “bins” of samples and operating on each to find the maximum sample, an average of all samples, or a min/max pair.

3. Render: In this step you’ll take your reduced samples and render them onscreen. This will commonly be performed with Quartz, but you could use any of Apple’s supported drawing frameworks. The nature of how you draw the data is dependent on how you reduced it. If you captured a min/max pair, you’ll draw a vertical line for each pair. If you captured an average or maximum sample value for each bin, you’ll find that using a Quartz Bezier path works well for drawing the waveforms.

You’ll find a starter project in the Chapter 8 directory called **THWaveformView_Starter**. You’ll build a `UIView` subclass capable of rendering the waveforms like the ones shown in [Figure 8.5](#). Let’s get started with the first step.

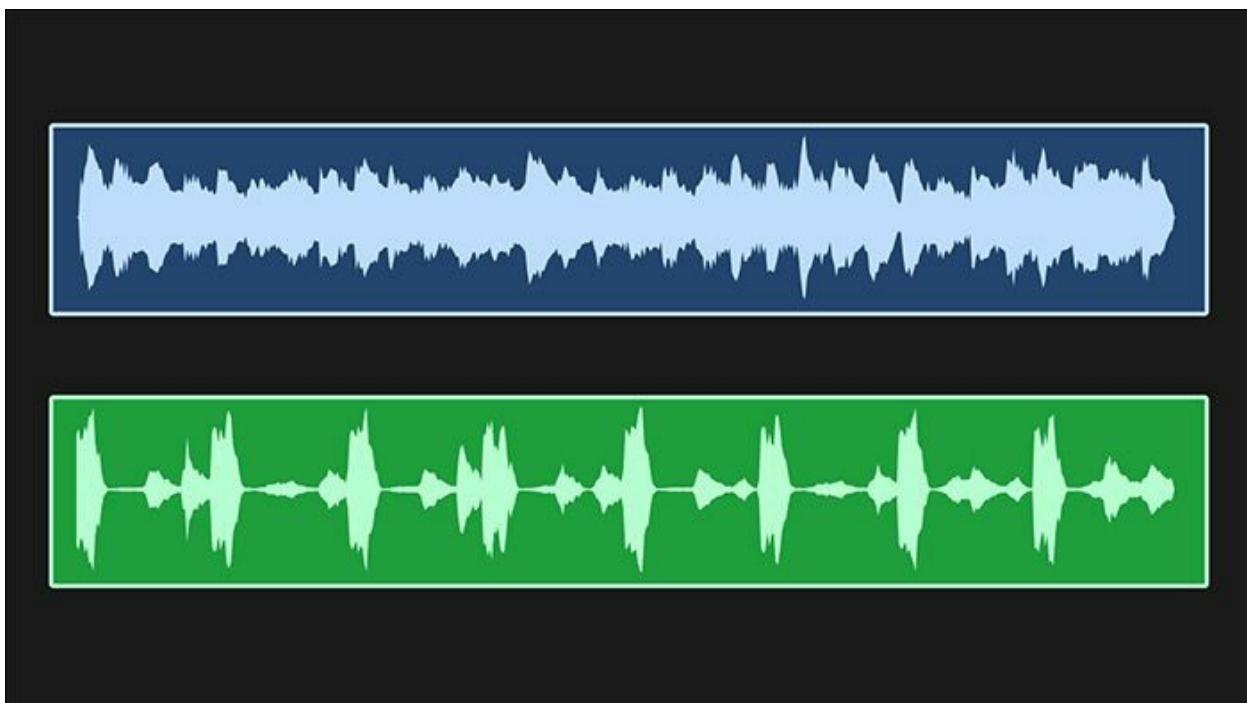


Figure 8.5 THWaveformView in action

Reading the Audio Samples

The first class to build is called `THSampleDataProvider`. This class uses an instance of `AVAssetReader` to read the audio samples from an `AVAsset` and return them as an `NSData` object. The interface for this class is shown in [Listing 8.1](#).

Listing 8.1 `THSampleDataProvider` Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>

typedef void(^THSampleDataCompletionBlock) (NSData *);

@interface THSampleDataProvider : NSObject

+ (void)loadAudioSamplesFromAsset:(AVAsset *)asset
                           completionBlock:
    (THSampleDataCompletionBlock)completionBlock;

@end
```

The interface to this class is straightforward with the primary point of interest being its `loadAudioSamplesFromAsset:completionBlock:` class method, which will be called to read the audio samples. Let's move on to the implementation of this class in [Listing 8.2](#).

Listing 8.2 `THSampleDataProvider` Implementation

[Click here to view code image](#)

```
#import "THSampleDataProvider.h"

@implementation THSampleDataProvider

+ (void)loadAudioSamplesFromAsset:(AVAsset *)asset
                           completionBlock:
    (THSampleDataCompletionBlock)completionBlock {

    NSString *tracks = @"tracks";

    [asset loadValuesAsynchronouslyForKeys:@[tracks]
completionHandler:^{ // 1
```

```

    AVKeyValueStatus status = [asset
statusOfValueForKey:tracks error:nil];

    NSData *sampleData = nil;

    if (status == AVKeyValueStatusLoaded)
        // 2
        sampleData = [self readAudioSamplesFromAsset:asset];
    }

    dispatch_async(dispatch_get_main_queue(),
        // 3
        completionBlock(sampleData);
    );
}];

}

+ (NSData *)readAudioSamplesFromAsset:(AVAsset *)asset {
    // To be implemented

    return nil;
}

@end

```

1. Begin by performing the standard asynchronous loading of the asset's required keys so that you don't block when accessing the asset's `tracks` property.
2. If the `tracks` key was successfully loaded, call the private `readAudioSamplesFromAsset:` method to read the samples from the asset's audio track.
3. Because the loading happens on an arbitrary background queue, you'll want to dispatch back to the main queue and invoke the completion block with the retrieved audio samples or `nil` if they could not be read.

Now we'll discuss the implementation of the `readAudioSamplesFromAsset:` method. In the previous chapter we discussed using `CMSampleBuffer` to access the video frames vended by an `AVCaptureVideoDataOutput` object. When you're working with uncompressed video data, you use the

`CMSampleBufferGetImageBuffer` function to retrieve the underlying `CVImageBufferRef` containing frame's pixels. When reading audio samples from an asset, you'll again be working with `CMSampleBuffer`, but in this case the underlying data will be provided in the form of a Core Media type called `CMBlockBuffer`. Block buffers are used to pass arbitrary bytes of data through the Core Media pipeline. Depending on what you intend to do with the audio samples, you can use a couple of ways to access the block buffer containing the audio data. You can get an *unretained* reference to the block buffer using the `CMSampleBufferGetDataBuffer` function, which works well in cases where you need to access the data but won't be further processing it. Alternatively, if you intend to process the audio data, such as passing it off to Core Audio, you can access the data as an `AudioBufferList` using the `CMSampleBufferGetAudioBufferListWithRetainedBlockBuf` function. This returns the data as a Core Audio `AudioBufferList` along with a retained `CMBlockBuffer` responsible for managing the lifetime of the samples contained in it. Because you'll be retrieving the samples and copying them to an `NSData`, you'll use the former approach to retrieve the audio data (see [Listing 8.3](#)).

Listing 8.3 Reading the Asset's Audio Samples

[Click here to view code image](#)

```
+ (NSData *)readAudioSamplesFromAsset:(AVAsset *)asset {  
    NSError *error = nil;  
  
    AVAssetReader *assetReader  
    = // 1  
        [ [AVAssetReader alloc] initWithAsset:asset error:&error];  
  
    if (!assetReader) {  
        NSLog(@"Error creating asset reader: %@", [error  
localizedDescription]);  
        return nil;  
    }  
  
    AVAssetTrack *track  
    = // 2  
        [ [asset tracksWithMediaType:AVMediaTypeAudio]  
firstObject];
```

```

NSDictionary *outputSettings =
@{                                         // 3
    AVFormatIDKey                  : @ (kAudioFormatLinearPCM),
    AVLinearPCMIsBigEndianKey     : @NO,
    AVLinearPCMIsFloatKey         : @NO,
    AVLinearPCMBitDepthKey        : @ (16)
};

AVAssetReaderTrackOutput *trackOutput
=                                         // 4
    [ [AVAssetReaderTrackOutput alloc] initWithTrack:track
                                                outputSettings:outputSett:

[assetReader addOutput:trackOutput];

[assetReader startReading];

NSMutableData *sampleData = [NSMutableData data];

while (assetReader.status == AVAssetReaderStatusReading) {

    CMSampleBufferRef sampleBuffer = [trackOutput
copyNextSampleBuffer];// 5

    if (sampleBuffer) {

        CMBlockBufferRef blockBufferRef
=                                         // 6
            CMSampleBufferGetDataBuffer(sampleBuffer);

        size_t length =
CMBlockBufferGetDataLength(blockBufferRef);
        SInt16 sampleBytes[length];

        CMBlockBufferCopyDataBytes(blockBufferRef,
7
                                         0,
                                         length,
                                         sampleBytes);

        [sampleData appendBytes:sampleBytes length:length];

        CMSampleBufferInvalidate(sampleBuffer);
8
        CFRelease(sampleBuffer);
    }
}

if (assetReader.status == AVAssetReaderStatusCompleted)
{
    // 9
}

```

```
        return sampleData;
    } else {
        NSLog(@"Failed to read audio samples from asset");
        return nil;
    }
}
```

1. Create a new instance of `AVAssetReader`, passing it the asset to read. If there was an error initializing the object, you'll log the error message to the console and return `nil`.
2. Retrieve the first audio track found in the asset. The audio files contained in the sample project contain only a single track, but it's best to always retrieve tracks by their desired media type.
3. Create an `NSDictionary` containing the decompression settings to use when reading the audio samples from the asset track. The samples need to be read in their uncompressed format, so you'll specify `kAudioFormatLinearPCM` as the format key. You'll also want to make sure they are read as 16-bit, signed integers in little-endian byte ordering. These settings suffice for the sample project, but you can find many additional keys in `AVAudioSettings.h` that provide you greater control over the format conversion.
4. Create a new instance of `AVAssetReaderTrackOutput` passing it the output settings you created in the previous step. You'll add it as an output on the `AVAssetReader` and call `startReading` to allow the asset reader to begin prefetching samples.
5. Begin each iteration of the loop by calling the track output's `copyNextSampleBuffer` method, which returns the next available sample buffer containing the audio samples.
6. The audio samples in the `CMSampleBuffer` will be contained in a type called `CMBlockBuffer`. You access this block buffer by using the `CMSampleBufferGetDataBuffer` function. You determine its length using the `CMBlockBufferGetDataLength` function and create an array of 16-bit signed integers to hold the audio samples.
7. Populate the array with the data contained in the `CMBlockBuffer` using the `CMBlockBufferCopyDataBytes` function, and append the array's contents to the `NSData` instance.

8. Use the `CMSampleBufferInvalidate` function to indicate that this sample buffer has been processed and invalidate it from further use. Additionally, you'll need to release the copied `CMSampleBuffer` to free its memory.
9. If the asset reader's status is equal to `AVAssetReaderStatusCompleted`, the data was successfully read, and you'll return the `NSData` containing the audio samples. If a failure occurred, you'll return `nil`.

The first step is complete, and you now have a means of successfully reading the audio samples from a variety of audio formats. The next step is to reduce the data into a useable form that can be drawn onscreen.

Reducing the Audio Samples

`THSampleDataProvider` will extract the complete set of samples from a given audio asset. Even with very small audio files, this can still result in hundreds of thousands of samples, which is far more than what is needed to draw onscreen. You need to define a way of filtering this set down to a collection of values that can ultimately be drawn onscreen. To perform this reduction you create an object called `THSampleDataFilter` that will perform this operation. The interface for this class is shown in [Listing 8.4](#).

Listing 8.4 `THSampleDataFilter` Interface

[Click here to view code image](#)

```

@interface THSampleDataFilter : NSObject
- (id)initWithData:(NSData *)sampleData;
- (NSArray *)filteredSamplesForSize:(CGSize)size;
@end

```

An instance of this class is initialized with an `NSData` containing the audio samples. It provides the `filteredSamplesForSize:` method that filters the data set down according to the specified size constraint.

There are two phases to process this data. In the first, you'll segment the samples into “bins,” finding the maximum sample in each. When all bins

have been processed, you'll apply a scaling factor to the samples relative to the size constraint passed to the `filteredSamplesForSize:` method. Let's look at the implementation of this class in [Listing 8.5](#).

Listing 8.5 THSampleDataFilter Implementation

[Click here to view code image](#)

```
#import "THSampleDataFilter.h"

@interface THSampleDataFilter ()
@property (nonatomic, strong) NSData *sampleData;
@end

@implementation THSampleDataFilter

- (id)initWithData:(NSData *)sampleData {
    self = [super init];
    if (self) {
        _sampleData = sampleData;
    }
    return self;
}

- (NSArray *)filteredSamplesForSize:(CGSize)size {
    NSMutableArray *filteredSamples = [[NSMutableArray alloc] init]; // 1
   NSUInteger sampleCount = self.sampleData.length / sizeof(SInt16);
    NSUInteger binSize = sampleCount / size.width;

    SInt16 *bytes = (SInt16 *)self.sampleData.bytes;

    SInt16 maxSample = 0;

    for (NSUInteger i = 0; i < sampleCount; i += binSize) {
        SInt16 sampleBin[binSize];

        for (NSUInteger j = 0; j < binSize; j++) // 2
            sampleBin[j] = CFSwapInt16LittleToHost(bytes[i + j]);
    }

    SInt16 value = [self maxValueInArray:sampleBin
        ofSize:binSize]; // 3
    [filteredSamples addObject:@(value)];
}
```

```

        if (value > maxSample)                                // 4
    {
        maxSample = value;
    }
}

CGFloat scaleFactor = (size.height / 2) / maxSample;

for (NSUInteger i = 0; i < filteredSamples.count; i++)      // 5
{
    filteredSamples[i] = @([filteredSamples[i] integerValue] *
scaleFactor);
}

return filteredSamples;
}

- (SInt16)maxValueInArray:(SInt16[])values ofSize:(NSUInteger)size
{
    SInt16 maxValue = 0;
    for (int i = 0; i < size; i++) {
        if (abs(values[i]) > maxValue) {
            maxValue = abs(values[i]);
        }
    }
    return maxValue;
}

@end

```

- 1.** Begin by creating an `NSMutableArray` to store the filtered array of audio samples. You also determine the total number of samples to process and calculate a “bin” size appropriate for the size constraint passed to the method. A bin contains a subset of samples to be filtered.
- 2.** Iterate through the complete set of audio samples, and in each iteration you’ll construct the bin of data to be processed. When working with audio samples, you should always keep byte ordering in mind, so you’ll use the `CFSwapInt16LittleToHost` function to ensure the samples are in the host’s native byte order.
- 3.** For each bin you find the maximum sample by calling the `maxValueInArray:` method. This method iterates through all the samples in the bin and finds the maximum absolute value. The resulting value is added to the `filteredSamples` array.
- 4.** As you iterate through all the audio samples, you calculate the

maximum value in the filtered values. This will be used to calculate a scaling factor to be applied to the filtered samples.

5. Prior to returning the filtered samples, you need to scale their values relative to the size constraint passed to this method. This results in an array of floating point values that can be rendered onscreen. When the values have been scaled, you return the array to the caller.

The `THSampleDataFilter` class is complete, and we're ready to discuss how to build the view to render the audio samples.

Rendering the Audio Samples

You'll build a `UIView` subclass to render the results. Let's begin by looking at the interface this class provides in [Listing 8.6](#).

Listing 8.6 THWaveformView Interface

[Click here to view code image](#)

```
@class AVAsset;

@interface THWaveformView : UIView

@property (strong, nonatomic) AVAsset *asset;
@property (strong, nonatomic) UIColor *waveColor;

@end
```

This view provides a simple interface that enables you to set the `AVAsset` and color in which its waveform should be drawn. Let's look at the implementation of this class starting in [Listing 8.7](#). Some of the `UIView` boilerplate code has been omitted from this listing. See the project source code for the complete implementation.

Listing 8.7 Implementing the `setAsset` Method

[Click here to view code image](#)

```
#import "THWaveformView.h"
#import "THSampleDataProvider.h"
#import "THSampleDataFilter.h"
#import <QuartzCore/QuartzCore.h>

static const CGFloat THWidthScaling = 0.95;
```

```

static const CGFloat THHeightScaling = 0.85;

@interface THWaveformView ()
@property (strong, nonatomic) THSampleDataFilter *filter;
@property (strong, nonatomic) UIActivityIndicatorView
*loadingView;
@end

@implementation THWaveformView

...

- (void)setAsset:(AVAsset *)asset {
    if (_asset != asset) {
        _asset = asset;

        [THSampleDataProvider
loadAudioSamplesFromAsset:self.asset           // 1
completionBlock:^(NSData
*sampleData) {

            self.filter
=           [[THSampleDataFilter alloc]
initWithData:sampleData];

            [self.loadingView
stopAnimating];                                // 3
            [self setNeedsDisplay];
        }];
    }
}

- (void)drawRect:(CGRect)rect {
    // To be implemented
}

@end

```

- 1.** Begin by loading the audio samples with the `THSampleDataProvider` class calling its `loadAudioSamplesFromAsset:completionBlock:` method.
- 2.** When the samples have been loaded, you construct a new instance of `THSampleDataFilter`, passing it the `NSData` containing the audio samples.
- 3.** You perform some housekeeping on the view by dismissing the view's

loading spinner and calling `setNeedsDisplay`, which causes the `drawRect:` method to be invoked.

Let's move on to the implementation of the `drawRect:` method so you can see how this data is drawn onscreen (see [Listing 8.8](#)).

Listing 8.8 Implementing the `drawRect:` Method

[Click here to view code image](#)

```
- (void)drawRect:(CGRect)rect {  
  
    CGContextRef context = UIGraphicsGetCurrentContext();  
  
    CGContextScaleCTM(context, THWidthScaling,  
    THHeightScaling); // 1  
  
    CGFloat xOffset = self.bounds.size.width -  
        (self.bounds.size.width * THWidthScaling);  
  
    CGFloat yOffset = self.bounds.size.height -  
        (self.bounds.size.height * THHeightScaling);  
  
    CGContextTranslateCTM(context, xOffset / 2, yOffset / 2);  
  
    NSArray *filteredSamples  
    = // 2  
        [self.filter filteredSamplesForSize:self.bounds.size];  
  
    CGFloat midY = CGRectGetMidY(rect);  
  
    CGMutablePathRef halfPath =  
    CGPathCreateMutable(); // 3  
    CGPathMoveToPoint(halfPath, NULL, 0.0f, midY);  
  
    for (NSUInteger i = 0; i < filteredSamples.count; i++) {  
        float sample = [filteredSamples[i] floatValue];  
        CGPathAddLineToPoint(halfPath, NULL, i, midY - sample);  
    }  
  
    CGPathAddLineToPoint(halfPath, NULL, filteredSamples.count,  
    midY);  
  
    CGMutablePathRef fullPath =  
    CGPathCreateMutable(); // 4  
    CGPathAddPath(fullPath, NULL, halfPath);  
  
    CGAffineTransform transform =  
    CGAffineTransformIdentity; // 5
```

```

        transform = CGAffineTransformTranslate(transform, 0,
CGRectGetHeight(rect));
        transform = CGAffineTransformScale(transform, 1.0, -1.0);
CGPathAddPath(fullPath, &transform, halfPath);

        CGContextAddPath(context,
fullPath);                                // 6
        CGContextSetFillColorWithColor(context,
self.waveColor.CGColor);
        CGContextDrawPath(context, kCGPathFill);

        CGPathRelease(halfPath);
7
        CGPathRelease(fullPath);
}

```

1. You want the waveform to be inset from the bounds of the view, so you begin by scaling the graphics context based on the defined height and width constants. You'll also calculate x and y offsets to translate the context to move the offsets appropriately within the scaled context.
2. You retrieve the filtered samples from the THSampleDataFilter instance, passing it the size of the view's bounds. In production code you'd likely want to move this retrieval out of the `drawRect:` method so you can better optimize when the samples are filtered, but this approach will suffice for the sample app.
3. Create a new `CGMutablePathRef`, which is used to draw the top half of the waveform's Bezier path. Iterate through the filtered samples, and for each you'll call `CGPathAddLineToPoint` to add a point to the path. You use the loop index as the x coordinate and the sample value as the y coordinate.
4. You create a second `CGMutablePathRef`, passing it the Bezier path constructed in step 4. You use this Bezier path to draw the complete waveform.
5. To draw the bottom half of the waveform, you apply a *translate* and *scale* transform to the top path. This results in the top path being flipped upside down, filling in the completed waveform.
6. Add the completed path to the graphics context, set the fill color according to the specified `waveColor`, and call `CGContextDrawPath(context, kCGPathFill)` to draw a

filled path into the context.

7. Whenever you create Quartz objects it's your responsibility to free their memory, so the last step is to call `CGPathRelease` on the path objects you created.

The application's view controller is set up to draw two views, as was shown previously in [Figure 8.5](#). You can open the application's view controller and experiment with the colors, or you can modify the views in the storyboard. You now have a nice reusable class you can use whenever you want to render waveforms in your application.

Advanced Capture Recording

At the end of the previous chapter we discussed rendering the `CVPixelBuffer` objects captured by `AVCaptureVideoDataOutput` as OpenGL ES textures. This is a very powerful capability that enables us to build fun and interesting applications. However, a big drawback of using `AVCaptureVideoDataOutput` is that you lose the convenience of `AVCaptureMovieFileOutput` to record the output. No matter how amazing your special effects may be, your app would be severely hamstrung if you were unable to record the output so you could share it with the world. In this section you learn how you can use `AVAssetWriter` to build a reusable class similar to `AVCaptureMovieFileOutput` to record the output from the advanced capture output classes.

We looked at using OpenGL ES in the previous chapter. We could reuse the `CubeKamera` app for this purpose, but instead of repeating ourselves, let's build a new camera app that uses the Core Image framework to process the video frames to apply real-time video effects.

You'll find a sample project called **KameraWriter_Starter** under the [Chapter 8](#) directory. `KameraWriter` is a video-recording application with real-time video effects. The user can toggle through the available effects from the filter selector at the top of the screen (see [Figure 8.6](#))



Figure 8.6 KameraWriter app (rotated)

Since iOS 7, the built-in Camera app enables you to apply filters when taking still images. These filters can be found in the Core Image framework and can be used to add real-time video effects to your apps when running on most current generation iOS devices. The details of Core Image are outside the scope of the book, but you'll have only minimal interaction with it in the course of developing this feature.

The first step is to build the camera controller. We'll build on the same infrastructure you used in the previous chapter to reduce the amount of boilerplate code required to get up and running.

Let's begin with the `THCameraController` interface shown in [Listing 8.9](#).

Listing 8.9 THCameraController Interface

[Click here to view code image](#)

```
#import "THImageTarget.h"
#import "THBaseCameraController.h"

@interface THCameraController : THBaseCameraController
- (void)startRecording;
- (void)stopRecording;
```

```
@property (nonatomic, getter = isRecording) BOOL recording;  
  
@property (weak, nonatomic) id <THImageTarget> imageTarget;  
  
@end
```

The interface provides methods to start and stop the recording, as well as to determine the recording status. It also provides an `imageTarget` property, which is the visual output for a Core Image `CIImage` object.

The next step is to begin building out the camera controller implementation, beginning with configuring the capture session outputs (see [Listing 8.10](#)).

Listing 8.10 Configuring the Session Outputs

[Click here to view code image](#)

```
#import "THCameraController.h"  
#import <AVFoundation/AVFoundation.h>  
  
@interface THCameraController ()  
<AVCaptureVideoDataOutputSampleBufferDelegate,  
 AVCaptureAudioDataOutputSampleBu:  
  
@property (strong, nonatomic) AVCaptureVideoDataOutput  
*videoDataOutput;  
@property (strong, nonatomic) AVCaptureAudioDataOutput  
*audioDataOutput;  
  
@end  
  
@implementation THCameraController  
- (BOOL)setupSessionOutputs:(NSError ***)error {  
    self.videoDataOutput = [[AVCaptureVideoDataOutput alloc]  
init]; // 1  
  
    NSDictionary *outputSettings =  
        @{@"(id)kCVPixelBufferPixelFormatTypeKey :  
        @ (kCVPixelFormatType_32BGRA)};  
  
    self.videoDataOutput.videoSettings = outputSettings;  
    self.videoDataOutput.alwaysDiscardsLateVideoFrames =  
NO; // 2  
  
    [self.videoDataOutput setSampleBufferDelegate:self  
queue:self.dispatchQueu
```

```

    if ([self.captureSession canAddOutput:self.videoDataOutput]) {
        [self.captureSession addOutput:self.videoDataOutput];
    } else {
        return NO;
    }

    self.audioDataOutput = [[AVCaptureAudioDataOutput alloc]
init]; // 3

    [self.audioDataOutput setSampleBufferDelegate:self
queue:self.dispatchQueue];

    if ([self.captureSession canAddOutput:self.audioDataOutput]) {
        [self.captureSession addOutput:self.audioDataOutput];
    } else {
        return NO;
    }

    return YES;
}

- (NSString *)sessionPreset // 4
{
    return AVCaptureSessionPresetMedium;
}

- (void)startRecording {
    // To be implemented
}

- (void)stopRecording {
    // To be implemented
}

#pragma mark - Delegate methods

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
fromConnection:(AVCaptureConnection *)connection {

    // To be implemented
}

@end

```

- 1. Begin by creating a new AVCaptureVideoDataOutput. You configure it the same way you did in the previous chapter by setting the**

format of its output to `kCVPixelFormatType_32BGRA`. This format works well when integrating with OpenGL ES or Core Image.

2. Set `alwaysDiscardsLateVideoFrames` to `NO`. Because you'll be recording the output, you'll typically want to capture all available frames. Setting this property to `NO` provides the delegate some additional time to process the sample buffers, but it can increase memory consumption. It's still critical that your processing of each sample buffer be performed as efficiently as possible to ensure real-time performance.
3. You also create a new instance of `AVCaptureAudioDataOutput`. This is the audio counterpart to `AVCaptureVideoDataOutput` and is used to capture audio samples from the audio device attached to the active `AVCaptureSession`.
4. You'll set the session preset to `AVCaptureSessionPresetMedium`. It's best to start with a modest preset setting when developing this application and then increase it to meet your needs after you know the functionality is working.

With the capture session's outputs configured, it's time to provide an implementation of the delegate callback method, as shown in [Listing 8.11](#).

Listing 8.11 Capture Output Delegate

[Click here to view code image](#)

```
- (void)captureOutput: (AVCaptureOutput *)captureOutput
didOutputSampleBuffer: (CMSampleBufferRef)sampleBuffer
fromConnection: (AVCaptureConnection *)connection {

    if (captureOutput == self.videoDataOutput)
        // 1

        CVPixelBufferRef imageBuffer
    =
        CMSampleBufferGetImageBuffer(sampleBuffer);
        // 2

        CIImage *sourceImage
    =
        [CIImage imageWithCVPixelBuffer:imageBuffer
options:nil];
        // 3
```

```
        [self.imageTarget setImage:sourceImage];
    }
}
```

1. The

captureOutput:didOutputSampleBuffer:fromConnection method is the delegate callback for both capture outputs. Currently, you are interested only in processing video samples, so you'll limit your processing to only those callbacks coming from the AVCaptureVideoDataOutput instance.

2. You get the underlying CVPixelBuffer from the sample buffer using the CMSampleBufferGetImageBuffer function.
3. You create a new CIImage from the CVPixelBuffer and pass it to the image target to be rendered onscreen.

The basic capture functionality is now complete, so you can build and run the application on your device to see the filters in action. You can tap the arrows next to the filter name to toggle through them. You now have real-time video filters being applied to your video frames, thanks to the Core Image framework!

Note

The sample app is using a single dispatch queue for both the AVCaptureVideoDataOutput and AVCaptureAudioDataOutput instances. This is sufficient for the purposes of the sample app, but if you are performing more elaborate processing of the data you may want to consider using a separate queue for each. Apple provides a sample app called RosyWriter (available on the ADC) that uses this approach. It also provides examples of some more advanced performance options for efficiently processing CMSampleBuffers.

The KameraWriter app looks good, and the basic functionality is working as intended, but you can't yet record the output. To resolve this you create an object called THMovieWriter that provides functionality similar to AVCaptureMovieFileOutput but using AVAssetWriter to perform the video encoding and file writing. Let's look at the interface for this object

in Listing 8.12.

Listing 8.12 THMovieWriter Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>

@protocol THMovieWriterDelegate <NSObject>
- (void)didWriteMovieAtURL:(NSURL *)outputURL;
@end

@interface THMovieWriter : NSObject

- (id)initWithVideoSettings:(NSDictionary *)
*)videoSettings           // 1
    audioSettings:(NSDictionary *)audioSettings
    dispatchQueue:(dispatch_queue_t)dispatchQueue;

- (void)startWriting;
- (void)stopWriting;
@property (nonatomic) BOOL isWriting;

@property (weak, nonatomic) id<THMovieWriterDelegate>
delegate;                      // 2

- (void)processSampleBuffer:
(CMSampleBufferRef)sampleBuffer;          // 3

@end
```

1. An instance of THMovieWriter is created with two dictionaries describing the configuration parameters for the underlying AVAssetWriter instance and a dispatch queue. It provides methods to start and stop the writing process and monitor its status as well.
2. This class defines a delegate protocol called THMovieWriterDelegate that indicates when a movie file has been written to disk; then the delegate can be notified and take the appropriate action.
3. The key method provided by THMovieWriter is processSampleBuffer:, which will be called continually as new samples are captured by the capture output objects.

Working with AVAssetWriter, even in simple cases, is fairly involved.

So we'll break the development of this class into some smaller chunks. Let's tackle the life-cycle methods first, as shown in [Listing 8.13](#).

Listing 8.13 **THMovieWriter** Life-Cycle Methods

[Click here to view code image](#)

```
#import "THMovieWriter.h"
#import <AVFoundation/AVFoundation.h>
#import "THContextManager.h"
#import "THFunctions.h"
#import "THPhotoFilters.h"
#import "THNotifications.h"

static NSString *const THVideoFilename = @"movie.mov";

@interface THMovieWriter ()

@property (strong, nonatomic) AVAssetWriter
*assetWriter; // 1
@property (strong, nonatomic) AVAssetWriterInput
*assetWriterVideoInput;
@property (strong, nonatomic) AVAssetWriterInput
*assetWriterAudioInput;
@property (strong, nonatomic)
    AVAssetWriterInputPixelBufferAdaptor
*assetWriterInputPixelBufferAdaptor;

@property (strong, nonatomic) dispatch_queue_t dispatchQueue;

@property (weak, nonatomic) CIContext *ciContext;
@property (nonatomic) CGColorSpaceRef colorSpace;
@property (strong, nonatomic) CIFilter *activeFilter;

@property (strong, nonatomic) NSDictionary *videoSettings;
@property (strong, nonatomic) NSDictionary *audioSettings;

@property (nonatomic) BOOL firstSample;

@end

@implementation THMovieWriter

- (id)initWithVideoSettings:(NSDictionary *)videoSettings
    audioSettings:(NSDictionary *)audioSettings
    dispatchQueue:(dispatch_queue_t)dispatchQueue {

    self = [super init];
    if (self) {
```

```

        _videoSettings = videoSettings;
        _audioSettings = audioSettings;
        _dispatchQueue = dispatchQueue;

        _ciContext = [THContextManager
sharedInstance].ciContext; // 2
        _colorSpace = CGColorSpaceCreateDeviceRGB();

        _activeFilter = [THPhotoFilters defaultFilter];
        _firstSample = YES;

        NSNotificationCenter *nc = [NSNotificationCenter
defaultCenter]; // 3
        [nc addObserver:self
            selector:@selector(filterChanged:)
            name:THFilterSelectionChangedNotification
            object:nil];
    }

    return self;
}

- (void)dealloc {
    CGColorSpaceRelease(_colorSpace);
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}

- (void)filterChanged:(NSNotification *)notification {
    self.activeFilter = [notification.object copy];
}

- (void)startWriting {
    // To be implemented
}

- (void)processSampleBuffer:(CMSampleBufferRef)sampleBuffer
    mediaType:(CMMediaType)mediaType {
    // To be implemented
}

- (void)stopWriting {
    // To be implemented
}

- (NSURL *)outputURL
{ // 4
    NSString *filePath =
        [NSTemporaryDirectory()
stringByAppendingPathComponent:THVideoFilename];
}

```

```

NSURL *url = [NSURL fileURLWithPath:filePath];
if ([[NSFileManager defaultManager]
fileExistsAtPath:url.path]) {
    [[NSFileManager defaultManager] removeItemAtURL:url
error:nil];
}
return url;
}

@end

```

1. In the class extension you create properties for the `AVAssetWriter` and its related objects. Each time the `startWriting` method is called, you'll build this graph of objects and maintain strong references to them for the duration of the writing session.
2. You get the shared Core Image context from the `THContextManager` object. This object is backed by OpenGL ES and is used to apply filters to the incoming video samples and render the result to a `CVPixelBuffer`.
3. You register as an observer of the `THFilterSelectionChangedNotification` notification. This notification is posted from the user interface when the user toggles through the list of available filters. The `filterChanged:` method is called whenever this notification is posted and updates the `activeFilter` property accordingly.
4. You define an `outputURL` method that will be used to configure the `AVAssetWriter` instance. This method defines an `NSURL` in the temp directory, deleting any previous files with the same name.

With the life-cycle configuration complete, it's time to move on to the implementation of the `startWriting` method, as shown in [Listing 8.14](#).

Listing 8.14 Setting Up the `AVAssetWriter` Graph

[Click here to view code image](#)

```

- (void)startWriting {
    dispatch_async(self.dispatchQueue,
    ^{
        NSError *error = nil;

```

```

        NSString *fileType = AVFileTypeQuickTimeMovie;
        self.assetWriter
    =
        [AVAssetWriter assetWriterWithURL:[self outputURL]
                               fileType:fileType
                               error:&error];
        if (!self.assetWriter || error) {
            NSString *formatString = @"Could not create
AVAssetWriter: %@";
            NSLog(@"%@", [NSString stringWithFormat:formatString,
error]);
            return;
        }

        self.assetWriterVideoInput
    =
        [[AVAssetWriterInput alloc]
initWithMediaType:AVMediaTypeVideo
                           outputSettings:self.videoSettings];
        self.assetWriterVideoInput.expectsMediaDataInRealTime =
YES;

        UIDeviceOrientation orientation = [UIDevice
currentDevice].orientation;
        self.assetWriterVideoInput.transform
    =
        // 4
        CGAffineTransformForDeviceOrientation(orientation);

        NSDictionary *attributes =
@{
    (id)kCVPixelBufferPixelFormatTypeKey :
@(kCVPixelFormatType_32BGRA),
    (id)kCVPixelBufferWidthKey :
self.videoSettings[AVVideoWidthKey],
    (id)kCVPixelBufferHeightKey :
self.videoSettings[AVVideoHeightKey],
    (id)kCVPixelFormatOpenGLGLESCompatibility :
(id)kCFBooleanTrue
};

        self.assetWriterInputPixelBufferAdaptor
    =
        // 6
        [[AVAssetWriterInputPixelBufferAdaptor alloc]
initWithAssetWriterInput:self.assetWriterVideoInput
sourcePixelBufferAttributes:attributes];

        if ([self.assetWriter
canAddInput:self.assetWriterVideoInput]) { // 7

```

```

        [self.assetWriter
addInput:self.assetWriterVideoInput];
    } else {
        NSLog(@"Unable to add video input.");
        return;
    }

    self.assetWriterAudioInput
=                                     // 8
        [[AVAssetWriterInput alloc]
initWithMediaType:AVMediaTypeAudio
                           outputSettings:self.audioSettings];
    self.assetWriterAudioInput.expectsMediaDataInRealTime =
YES;

    if ([self.assetWriter
canAddInput:self.assetWriterAudioInput]) {      // 9
        [self.assetWriter
addInput:self.assetWriterAudioInput];
    } else {
        NSLog(@"Unable to add audio input.");
        return;
    }

    self.isWriting =
YES;                                         // 10
    self.firstSample = YES;
} );
}

```

- 1.** To prevent blocking when the user taps the Record button, you asynchronously dispatch to the `dispatchQueue` to perform the `AVAssetWriter` object graph set up.
- 2.** Create a new instance of `AVAssetWriter`, passing it the output URL where it should write the file, a file type constant, and an `NSError`. If any errors were encountered creating this object, you log the error message to the console and return.
- 3.** You create a new `AVAssetWriterInput` to append the samples coming from the `AVCaptureVideoDataOutput`. You pass its initializer a media type of `AVMediaTypeVideo` and the video settings with which the `THMovieWriter` was created. Set its `expectsMediaDataInRealTime` property to `YES` to indicate this input should be optimized for real-time use.

4. The application's user interface is locked to portrait orientation, but you want to allow capture in whatever orientation the user is holding the device. You determine the user interface orientation and use the `THTransformForDeviceOrientation` function to set an appropriate transform on the input. The orientation remains fixed at this setting for the duration of the writing session.
5. Define an `NSDictionary` of attributes used to configure the `AVAssetWriterInputPixelBufferAdaptor` you create in the next step. To ensure maximum efficiency, the values in this dictionary should correspond to the source pixel format that you used when configuring the `AVCaptureVideoDataOutput`.
6. You create a new `AVAssetWriterInputPixelBufferAdaptor`, passing it the attributes you created in the previous step. This object provides an optimized `CVPixelBufferPool` you can use for creating `CVPixelBuffer` objects for rendering the filtered video frames.
7. Add the video input to the asset writer. If the input could not be added, you log the error to the console and return.
8. Create an `AVAssetWriterInput` used to append the samples coming from the `AVCaptureAudioDataOutput`. You pass its initializer a media type of `AVMediaTypeAudio` and the audio settings with which the `THMovieWriter` was created. Set its `expectsMediaDataInRealTime` property to YES to indicate this input should be optimized for real-time use.
9. Add the audio input to the asset writer. If the input could not be added, log the error to the console and return.
10. Set the `isWriting` and `firstSample` properties to YES so you can begin appending samples.

Next, let's look at the implementation of the `processSampleBuffer:` method where you'll append the `CMSampleBuffer` objects as they are captured by the capture outputs (see [Listing 8.15](#)).

Listing 8.15 Processing the Sample Buffers

[Click here to view code image](#)

```

- (void)processSampleBuffer:(CMSampleBufferRef)sampleBuffer {

    if (!self.isWriting) {
        return;
    }

    CMFormatDescriptionRef formatDesc
    = CMSampleBufferGetFormatDescription(sampleBuffer); // 1

    CMMediaType mediaType =
    CMFormatDescriptionGetMediaType(formatDesc);

    if (mediaType == kCMMediaType_Video) {

        CMTime timestamp =
            CMSampleBufferGetPresentationTimeStamp(sampleBuffer);

        if (self.firstSample) // 2
            if ([self.assetWriter startWriting]) {
                [self.assetWriter
startSessionAtSourceTime:timestamp];
            } else {
                NSLog(@"Failed to start writing.");
            }
        self.firstSample = NO;
    }

    CVPixelBufferRef outputRenderBuffer = NULL;

    CVPixelBufferPoolRef pixelBufferPool =
        self.assetWriterInputPixelBufferAdaptor.pixelBufferPoo:

        OSStatus err =
    CVPixelBufferPoolCreatePixelBuffer(NULL, // 3
                                         pixelBuf:
                                         &outputRe
        if (err) {
            NSLog(@"Unable to obtain a pixel buffer from the
pool.");
            return;
        }

        CVPixelBufferRef imageBuffer
    = CMSampleBufferGetImageBuffer(sampleBuffer); // 4

        CIImage *sourceImage = [CIImage
imageWithCVPixelBuffer:imageBuffer

```

```

options:nil];

    [self.activeFilter setValue:sourceImage
forKey:kCIInputImageKey];

    CIImage *filteredImage = self.activeFilter.outputImage;

    if (!filteredImage) {
        filteredImage = sourceImage;
    }

    [self.ciContext
render:filteredImage // 5
        toCVPixelBuffer:outputRenderBuffer
            bounds:filteredImage.extent
colorSpace:self.colorSpace];

    if (self.assetWriterVideoInput.readyForMoreMediaData)
    { // 6
        if (!![self.assetWriterInputPixelBufferAdaptor
            appendPixelBuffer:outputRenderBuffer
            withPresentationTime:timestamp]) {
            NSLog(@"Error appending pixel buffer.");
        }
    }

    CVPixelBufferRelease(outputRenderBuffer);

}

else if (!self.firstSample && mediaType == kCMMediaType_Audio)
{
    // 7
    if (self.assetWriterAudioInput.isReadyForMoreMediaData) {
        if (![self.assetWriterAudioInput
appendSampleBuffer:sampleBuffer]) {
            NSLog(@"Error appending audio sample buffer.");
        }
    }
}

}

```

1. This method is processing both audio and video samples, so you need to determine the sample's media type so you can append to the correct writer input. You ask the sample buffer for its `CMFormatDescription` and use the `CMFormatDescriptionGetMediaType` to determine its media

type.

2. If processing the first video sample encountered after the user tapped the Record button, you start a new writing session by calling the asset writer's `startWriting` and `startSessionAtSourceTime:` methods, passing the sample's presentation time as the source time.
3. You create an empty `CVPixelBuffer` from the pixel buffer adaptor's pool. Use this pixel buffer to render the output of the filtered video frame.
4. Get the `CVPixelBuffer` for the current video sample using the `CMSampleBufferGetImageBuffer` function. You create a new `CIImage` from the pixel buffer and set it as the active filter's `kCIInputImageKey` value. You then ask the filter for its output image, which returns a `CIImage` object encapsulating the `CIFilter` operations. If for any reason the `filteredImage` is `nil`, you set the `CIImage` reference back to the original `sourceImage`.
5. Render the output of the filtered `CIImage` into the `CVPixelBuffer` you created in step 3.
6. If the video input's `readyForMoreMediaData` property is YES, you append the pixel buffer to the `AVAssetWriterPixelBufferAdaptor` along with the current sample's presentation time. You're finished processing the current video sample, so release the pixel buffer by calling the `CVPixelBufferRelease` function.
7. If the first sample has been processed and the current `CMSampleBuffer` is an audio sample, you ask the audio `AVAssetWriterInput` if it is ready for more data. If so, you append it to the input.

The `processSampleBuffer:` method is complete, so the last bit of functionality to implement is the `stopWriting` method (see [Listing 8.16](#)).

Listing 8.16 Finishing the Writing Session

[Click here to view code image](#)

```
- (void)stopWriting {  
    self.isWriting =
```

```

NO; // 1

dispatch_async(self.dispatchQueue, ^{
    [self.assetWriter
finishWritingWithCompletionHandler:^( // 2

        if (self.assetWriter.status ==
AVAssetWriterStatusCompleted) {
            dispatch_async(dispatch_get_main_queue(), // 3
{
                NSURL *fileURL = [self.assetWriter outputURL];
                [self.delegate didWriteMovieAtURL:fileURL];
            });
        } else {
            NSLog(@"Failed to write movie: %@", self.assetWriter.error);
        }
    }];
});
}

```

- 1.** Set the `isWriting` flag to `NO` so that no more samples will be processed in the `processSampleBuffer:mediaType:` method.
- 2.** Call `finishWritingWithCompletionHandler:` to finalize the writing session and close the file on disk.
- 3.** Determine the asset writer's status. If it equals `AVAssetWriterStatusCompleted`, indicating the file was successfully written, you dispatch back to the main thread invoking the delegate's `didWriteMovieAtURL:` method. For any other status, you log a message with the asset writer's error to the console.

The `THMovieWriter` class is complete and the one remaining step is to integrate it into the `THCameraController` (see [Listing 8.17](#)).

Listing 8.17 Using the `THMovieWriter`

[Click here to view code image](#)

```

#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>
#import "THMovieWriter.h"
#import <AssetsLibrary/AssetsLibrary.h>

@interface THCameraController ()

```

```

<AVCaptureVideoDataOutputSampleBufferDelegate,
    AVCaptureAudioDataOutputSampleBu:
THMovieWriterDelegate@property (strong, nonatomic) THMovieWriter
*movieWriter; // 1

@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError **)error {

    // AVCaptureVideoDataOutput and AVCaptureAudioDataOutput set
    up and
    // configuration previously covered in Listing 8.x

    NSString *fileType = AVFileTypeQuickTimeMovie;

    NSDictionary *videoSettings
= // 2
    [self.videoDataOutput
        recommendedVideoSettingsForAssetWriterWithOutputFileType];

    NSDictionary *audioSettings =
    [self.audioDataOutput
        recommendedAudioSettingsForAssetWriterWithOutputFileType];

    self.movieWriter
= // 3
    [[THMovieWriter alloc] initWithVideoSettings:videoSettings
        audioSettings:audioSettings
        dispatchQueue:self.dispatchQueue];
    self.movieWriter.delegate = self;

    return YES;
}

- (NSString *)sessionPreset {
    return AVCaptureSessionPreset1280x720;
}

- (void)startRecording // 4
{
    [self.movieWriter startWriting];
}

```

```

        self.recording = YES;
    }

- (void)stopRecording {
    [self.movieWriter stopWriting];
    self.recording = NO;
}

#pragma mark - Delegate methods

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
fromConnection:(AVCaptureConnection *)connection {
    [self.movieWriter
processSampleBuffer:sampleBuffer]; // 5

    if (captureOutput == self.videoDataOutput) {

        CVPixelBufferRef imageBuffer =
        CMSampleBufferGetImageBuffer(sampleBuffer);

        CIImage *sourceImage =
        [CIImage imageWithCVPixelBuffer:imageBuffer
options:nil];

        [self.imageTarget setImage:sourceImage];
    }
}

- (void)didWriteMovieAtURL:(NSURL *)outputURL
// 6
{
    ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];

    if ([library
videoAtPathIsCompatibleWithSavedPhotosAlbum:outputURL]) {

        ALAssetsLibraryWriteVideoCompletionBlock completionBlock;

        completionBlock = ^(NSURL *assetURL, NSError *error){
            if (error) {
                [self.delegate
assetLibraryWriteFailedWithError:error];
            }
        };

        [library writeVideoAtPathToSavedPhotosAlbum:outputURL
                                         completionBlock:completionBlock];
    }
}

```

```
}
```

```
@end
```

1. Create a new property to store a strong reference to the `THMovieWriter` object. You also have the class adopt the `THMovieWriterDelegate` protocol.
2. iOS 7 introduced a couple convenient methods for easily building the dictionary of recommended audio and video settings for a given file type needed for configuring an `AVAssetWriter`. You call these methods and build the dictionaries you'll pass to the `THMovieWriter`.
3. Create a new instance of `THMovieWriter`, passing it the configuration dictionaries and a reference to the controller's `dispatchQueue`, which is defined in the superclass (`THBaseCameraController`). Mark the controller as the movie writer's delegate.
4. Implement the `startRecording` and `stopRecording` methods, calling the corresponding methods on the `THMovieWriter`. You also update the recording status as needed in each.
5. Call the movie writer's `processSampleBuffer:` method, passing it the current `CMSampleBuffer` and media type.
6. Implement the delegate protocol's `didWriteMovieAtURL:` method. You write the newly created movie to the user's Photos library using the `ALAssetsLibrary` framework as you have done previously in this book.

Now it's time to run the application and record your video. Start the application and tap the Record button. While the recording is in progress, you can tap through the available filters. Tap the Record button again to stop the recording. You can switch over to the iOS Photos app and see your video in all its filtered glory!

Summary

In this chapter you learned about the powerful capabilities provided by `AVAssetReader` and `AVAssetWriter`. These classes are the lowest

level ways of processing media with AV Foundation and play a vitally important role in many advanced use cases. Working with media at the CMSampleBuffer level exposes you to a certain amount of complexity, but also enables you to implement advanced features that couldn't be achieved using the framework's high-level functionality alone. Mastering AVAssetReader and AVAssetWriter can take some time, but the topics and sample projects covered in this chapter provide a useful foundation for you to build on in your own applications.

Challenge

An important part of learning to use AVAssetReader and AVAssetWriter is to gain a solid understanding of what kind of features you can build with these classes. This chapter covered some important and common use cases, but it's really just scratching the surface. Using the knowledge you gained in this chapter, explore some of Apple's sample code and other resources available on the Apple Developer Center:

- StopNGo for iOS shows you how to build a stop-motion camera app.
- “Sample Photo Editing Extension” demonstrates building an iOS 8 editing extension.
- Technical Note TN2310 explains how to work with timecode using AVAssetReader and AVAssetWriter.
- “Writing Subtitles to a Movie from the Command Line for OS X.”

III: Media Creation and Editing

[9 Composing and Editing Media](#)

[10 Mixing Audio](#)

[11 Building Video Transitions](#)

[12 Layering Animated Content](#)

9. Composing and Editing Media

There has never been a better time to be producing digital media. Professional-grade digital SLR cameras are well within the reach of the aspiring auteur, affordable studio-quality digital audio interfaces have enabled musicians around the world to produce music rivaling the sonic quality of commercial studios, and iOS and other portable devices enable you to produce digital audio and video on the go. Having the capability of capturing high-quality digital media is truly fun and exciting. However, its only part of the production process. Arguably, the most important part of any media production is work performed in post-production editing.

Editing is the process of selecting, arranging, and manipulating the best shots and takes to provide coherence and continuity to your work. Great editing is critical to the success of any professional film or audio production, but it matters to *all* media regardless of scale. An unedited podcast will fail to keep the listener's interest, a video training course containing countless mistakes can leave a student confused, and posting hours of unedited footage of your family's Hawaiian vacation is unlikely to garner many "Likes" on Facebook. AV Foundation provides you with the APIs to build nonlinear, nondestructive editing tools and applications. The term nonlinear means you are free to combine, split, trim, overlap, and rearrange media clips in any order until they are in the precise sequence you want. The best part is that all these edits are nondestructive, meaning your original source media remains unaltered, freeing you to edit with impunity. In this chapter we'll explore one of the most essential steps in the editing process, which is to compose media from multiple sources.

Composing Media

Imagine for a moment that you recently took a trip to San Francisco. While you were there you captured some great footage at Golden Gate Park, the Japanese Tea Gardens, and Fisherman's Wharf. You'd like to combine the best shots from those videos, along with a fitting soundtrack, into a single movie you could share with your friends and family (see [Figure 9.1](#)).

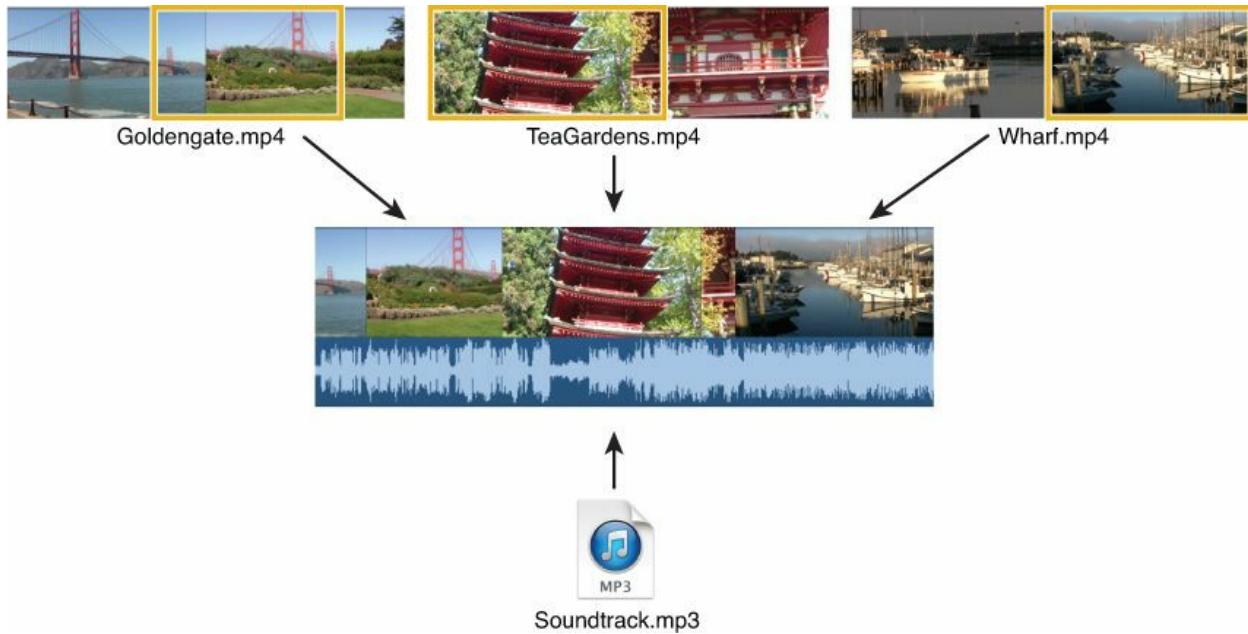


Figure 9.1 Composing media

To perform this task you need to extract the relevant segments from the source media and compose them into a temporal arrangement. AV Foundation provides precisely this functionality, enabling you to easily combine multiple audio and video assets into a new, composite asset. [Figure 9.2](#) provides an illustration of the core composition classes provided by the framework.

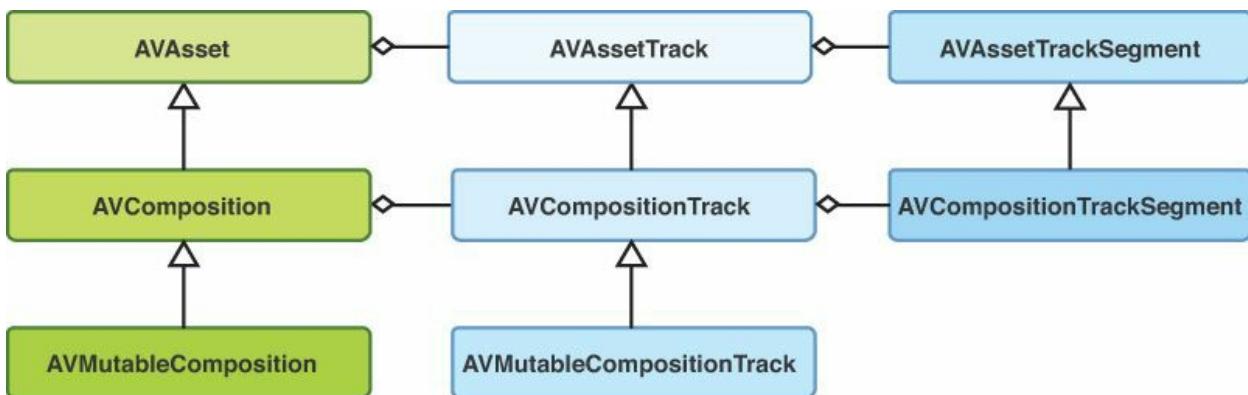


Figure 9.2 AV Foundation composition classes

AV Foundation's composition capabilities are built around a subclass of `AVAsset` called `AVComposition`. A composition is used to combine other media assets into a custom temporal arrangement so they can be presented or processed as a single media item. Just like an `AVAsset`, a composition acts as a container for one or more tracks of media of a given

type. The tracks in an `AVComposition` are a subclass of `AVAssetTrack` called `AVCompositionTrack`. A composition track itself is composed of one or more media segments, defined by the `AVCompositionTrackSegment` class, which represent the actual media regions in the composition. It's often easier to view the relationships between these objects in a manner similar to how they would be depicted in an editing tool such as iMovie or Logic Pro X (see [Figure 9.3](#)).

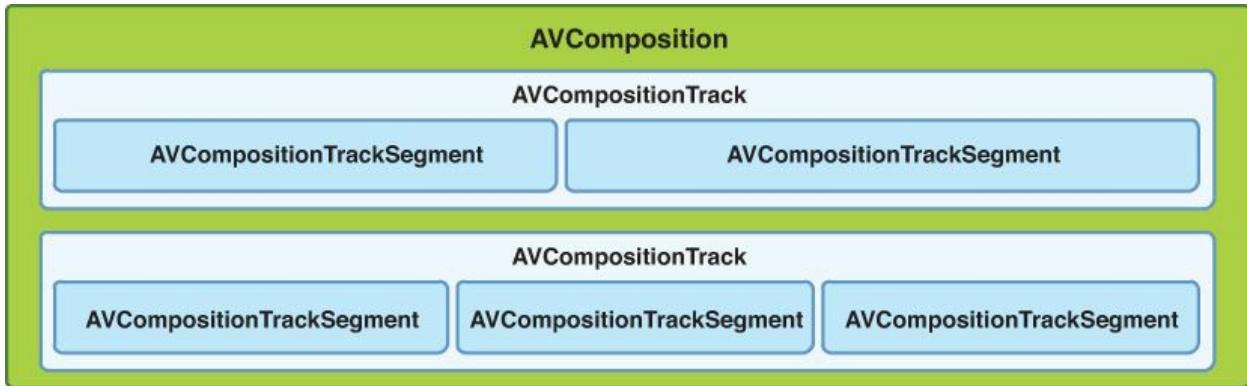


Figure 9.3 AV Foundation composition arrangement

As `AVComposition` extends `AVAsset`, it can be used in all the same scenarios where a normal asset would be used, such as playback, image extraction, or export. However, a composition should be considered in slightly more abstract terms. Whereas an `AVAsset` has a direct one-to-one mapping to a particular media file, a composition is more like a set of instructions for how multiple sources of media are to be presented or processed collectively. As such, they're very lightweight, transient objects that you'll create and dispose of frequently.

Note

A commonly asked question is, how do you save a composition? The short answer is, you don't. `AVComposition` and its related classes don't adopt the `NSCoding` protocol, so you can't simply archive a composition's in-memory state to disk. If you're building an audio or video editing application that requires the capability of saving a project file, you'll need to develop your own custom data model classes to persist this state.

An `AVComposition` and `AVCompositionTrack` are immutable objects

providing a read-only perspective of the resource. These objects provide a suitable interface to pass to other parts of your application for playback or processing. However, when building your own compositions you'll be working with their mutable subclasses found in `AVMutableComposition` and `AVMutableCompositionTrack`. These objects provide the interfaces needed to manipulate tracks and track segments so you can create any temporal arrangement you require.

To build custom compositions, you need to specify time ranges within the source media that you will add to your composition and the time location in each track where the clip should be added. Before we get into the mechanics of building a composition, we first need to discuss how time and time ranges are represented in the framework.

Working with Time

We briefly discussed working with time in [Chapter 4, “Playing Video,”](#) when discussing video playback. To effectively build compositions, it is essential to have a solid understanding of working with time and time ranges, so let's explore this topic further.

CMTIME

It's common for developers to think of representing time as a floating-point type. Anyone who has spent time developing for Apple platforms has certainly used `NSTimeInterval`, which is simply a `double` type, to represent time in a variety of scenarios. In fact, AV Foundation itself uses this type when working with time in `AVAudioPlayer` and `AVAudioRecorder`. Although there are many general programming scenarios where a floating-point representation of time suffices, its inherent imprecision makes it impractical to meet the advanced needs of working with timed media. For instance, a single rounding error can lead to a dropped frame or cause an audio dropout. Instead, Apple represents time as a rational number defined by the Core Media framework's `CMTIME` data type. This is a C struct type defined as follows:

[Click here to view code image](#)

```
typedef struct {
    CMTIMEValue value;
    CMTIMEScale timescale;
```

```
    CMTIMEFLAGS flags;
    CMTIMEPOCH epoch;
} CMTIME;
```

The three components of this struct that are most relevant are the `value`, `timescale`, and `flags`. A `CMTIMEVALUE` and `CMTIMESCALE` are 64-bit and 32-bit signed integers, respectively, and form the fractional component of a `CMTIME`. The `CMTIMEFLAGS` is a bit mask used to denote special states the time may have, such as if it's valid, indefinite, or if any rounding of its value has occurred. Instances of `CMTIME` can mark specific points in time or can be used to denote durations of time.

There are a number of ways to create an instance of `CMTIME`, but the most common is to use the `CMTIMEMAKE` function, which takes a 64-bit `value` and a 32-bit `timescale`. For instance, creating a `CMTIME` representing 3 seconds could be expressed in a variety of ways.

[Click here to view code image](#)

```
CMTIME t1 = CMTIMEMAKE(3, 1);
CMTIME t2 = CMTIMEMAKE(1800, 600);
CMTIME t3 = CMTIMEMAKE(3000, 1000);
CMTIME t4 = CMTIMEMAKE(132300, 44100);
```

Using the `CMTIMESHOW` function to print these values to the console would result in the following output.

[Click here to view code image](#)

```
CMTIMESHOW(t1); // --> {3/1 = 3.000}
CMTIMESHOW(t2); // --> {1800/600 = 3.000}
CMTIMESHOW(t3); // --> {3000/1000 = 3.000}
CMTIMESHOW(t4); // --> {132300/44100 = 3.000}
```

Note

A common timescale you'll encounter when working with video is 600, because this is a common multiple of the most frequently used video frame rates of 24, 25, and 30FPS. The timescale most commonly used with audio data is its sampling rate, such as 44,100 (44.1kHz) or 48,000 (48kHz).

In addition to the functions you use to create instances of `CMTIME`, you'll find a variety of functions defined in `CMTIME.h` enabling you to perform a

variety of time calculations. For instance, a frequent need you'll have when building compositions is to add and subtract times. These calculations can be easily performed using the `CMTimeAdd` and `CMTimeSubtract` functions.

[Click here to view code image](#)

```
CMTIME time1 = CMTIMEMake(5, 1);
CMTIME time2 = CMTIMEMake(3, 1);

CMTIME result;

result = CMTIMEAdd(time1, time2);
CMTIMEShow(result); // --> {8/1 = 8.000}

result = CMTIMESubtract(time1, time2);
CMTIMEShow(result); // --> {2/1 = 2.000}
```

Generally, you should use the functions provided by the Core Media framework when performing time calculations because they take into account things such as differing time scales and flags. However, in some *limited* cases even some elementary school math can be used to good effect. For instance, you can create new time values by manipulating the timescale of another.

[Click here to view code image](#)

```
CMTIME time = CMTIMEMake(5000, 1000);

CMTIME doubledTime = CMTIMEMake(time.value, time.timescale / 2);
CMTIMEShow(doubledTime); // --> {5000/500 = 10.000}

CMTIME halvedTime = CMTIMEMake(time.value, time.timescale * 2);
CMTIMEShow(halvedTime); // --> {5000/2000 = 2.500}
```

The preceding examples show some of the more common `CMTIME` functions, but I encourage you to view the documentation for `CMTIME` to see the other ways you can create, manipulate, and compare `CMTIME` values. In addition to the variety of functions, you'll find a number of constants and macros that you'll use frequently.

CMTIMERange

The Core Media framework additionally provides a data type, called `CMTIMERANGE`, for working with time ranges, which plays an important role in the editing APIs. A `CMTIMERANGE` is composed of two `CMTIME` values: the first defining the time range's starting time and the second defining the

time range's duration.

```
typedef struct {
    CMTime start;
    CMTime duration;
} CMTIMERange;
```

One way of creating a `CMTIMERange` is with the `CMTIMERangeMake` function, which takes a `CMTime` value defining the time range's starting time and a second `CMTime` to indicate its duration. For instance, if you wanted to create a time range that starts at 5 seconds along the timeline and is 5 seconds in duration, you could create this time range as follows:

[Click here to view code image](#)

```
CMTime fiveSecondsTime = CMTimeMake(5, 1);
CMTIMERange timeRange = CMTIMERangeMake(fiveSecondsTime,
fiveSecondsTime);
CMTIMERangeShow(timeRange); // --> {{5/1 = 5.000}, {5/1 = 5.000}}
```

Another way of creating a time range is with the `CMTIMERangeFromTimeToTime` function. This function creates a `CMTIMERange` by passing it `CMTime` values marking the range's starting and ending times. For instance, an alternate way of writing the previous example would be as follows:

[Click here to view code image](#)

```
CMTime fiveSeconds = CMTimeMake(5, 1);
CMTime tenSeconds = CMTimeMake(10, 1);
CMTIMERange timeRange = CMTIMERangeFromTimeToTime(fiveSeconds,
tenSeconds);
CMTIMERangeShow(timeRange); // --> {{5/1 = 5.000}, {5/1 = 5.000}}
```

The `CMTIMERange.h` header provides a number of useful functions to perform calculations and comparisons on time ranges. For instance, [Figure 9.4](#) shows two overlapping time ranges, both 5 seconds in duration, but with different starting times.

Start:kCMTimeZero

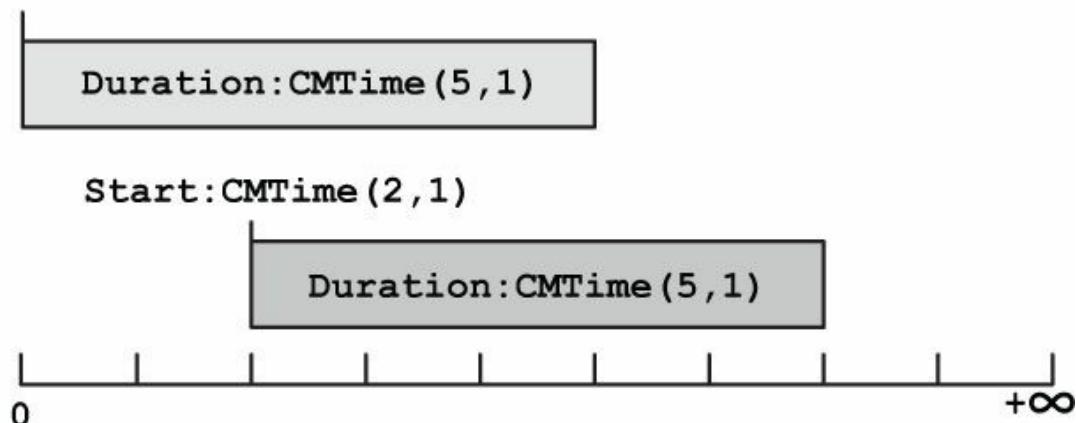


Figure 9.4 Overlapping time ranges

If you wanted to create a time range where the two intersect, or if you wanted to get a union of both time ranges, you could perform the following operations.

[Click here to view code image](#)

```
CMTIMERange range1 = CMTIMERangeMake(kCMTimeZero, CMTIMEMake(5, 1));
CMTIMERange range2 = CMTIMERangeMake(CMTIMEMake(2, 1),
CMTIMEMake(5, 1));

CMTIMERange intersectionRange = CMTIMERangeGetIntersection(range1,
range2);
CMTIMERangeShow(intersectionRange); // --> {{2/1 = 2.000}, {3/1 =
3.000} }

CMTIMERange unionRange = CMTIMERangeGetUnion(range1, range2);
CMTIMERangeShow(unionRange); // --> {{0/1 = 0.000}, {7/1 = 7.000}}
```

Learning to think of time as a rational number can seem odd at first and is often a point of confusion for those new to the framework's editing features. However, it's an essential skill you need to grasp when building custom compositions. The good news is that using CMTIME and CMTIMERange will quickly become second nature. You'll be making heavy use of these data types over the next few chapters, so you'll be comfortable using them shortly.

Basic Recipe

Armed with a good understanding of using CMTIME and CMTIMERange, let's move on and discuss building a composition similar to the one shown earlier in [Figure 9.1](#).

Building a composition requires that you have one or more source AVAsset objects at your disposal. Before we begin with the composition example, it's important to make a quick point regarding creating and preparing assets. Most of the examples throughout the book have created instances of AVAsset using its `assetWithURL:` class method. Under the hood this creates an instance of its subclass AVURLAsset. When creating assets for use in a composition, you should directly instantiate an AVURLAsset using its `URLAssetWithURL:options:` method. The `options:` argument provides you the opportunity to customize how the asset is initialized by passing an NSDictionary containing one or more initialization options. Let's look at a quick example of loading an MP4 video contained in the application bundle.

[Click here to view code image](#)

```
NSURL *url =
    [[NSBundle mainBundle] URLForResource:@"video"
withExtension:@"mp4"];

NSDictionary *options =
@{AVURLAssetPreferPreciseDurationAndTimingKey : @YES};
AVAsset *asset = [AVURLAsset URLAssetWithURL:url options:options];

// Asset "keys" to load
NSArray *keys = @[@"tracks", @"duration", @"commonMetadata"];

[asset loadValuesAsynchronouslyForKeys:keys completionHandler:^{
    // Validate loaded status of keys
}];
```

The framework defines an option called `AVURLAssetPreferPreciseDurationAndTimingKey`. Passing this option with a value of `@YES` ensures that precise duration and timing is computed when the asset's properties are loaded using the `AVAsynchronousKeyValueLoading` protocol. Some additional overhead can be added to the loading process when using this option, but it ensures the asset is in an appropriate state for editing purposes.

Let's move on and begin building the composition. This composition will take the first 5 seconds of two video clips and arrange them sequentially along the composition's video track. The composition will also provide an audio soundtrack from an MP3 file to accompany the video.

[Click here to view code image](#)

```

AVAsset *goldenGateAsset = // prepared golden gate asset
AVAsset *teaGardenAsset = // prepared tea garden asset
AVAsset *soundtrackAsset = // prepared sound track asset

AVMutableComposition *composition = [AVMutableComposition
composition];

// Video Track
AVMutableCompositionTrack *videoTrack =
[composition addMutableTrackWithMediaType:AVMediaTypeVideo
preferredTrackID:kCMPersistentTrackID
// Audio Track
AVMutableCompositionTrack *audioTrack =
[composition addMutableTrackWithMediaType:AVMediaTypeAudio
preferredTrackID:kCMPersistentTrackID

```

The preceding code example creates an `AVMutableComposition` and adds two track objects to it using its `addMutableTrackWithMediaType:preferredTrackID:` method. When creating a composition track you must indicate the media type it is capable of holding, as well as provide a track identifier. The `preferredTrackID:` argument takes a `CMPersistentTrackID`, which is a 32-bit integer value. You're free to pass an identifier of your choosing, which can be useful when you need to refer back to the track at a later time, but you'll commonly pass a constant value of `kCMPersistentTrackID_Invalid`. This oddly named constant means you'll delegate the responsibility of creating an appropriate track ID to the framework, which results in identifiers being generated in the range of 1..n. At this point you have a composition that looks like the one shown in [Figure 9.5](#). The composition is effectively, but it provides the necessary track arrangement for you to begin adding media segments to the individual tracks.



Figure 9.5 Composition state

The next step is to insert the individual media segments into the composition's tracks, as shown in the following code example. This code is a bit more involved, so it will be followed by an annotated breakdown discussing the relevant steps.

[Click here to view code image](#)

```

// The "insertion cursor" time
CMTime cursorTime =
kCMTimeZero;                                // 1

CMTime videoDuration = CMTimeMake(5,
1);                                         // 2
CMTimeRange videoTimeRange = CMTimeRangeMake(kCMTimeZero,
videoDuration);

AVAssetTrack *assetTrack;

// Extract and insert Golden Gate Segment
assetTrack
=                                              // ,
3
[[goldenGateAsset tracksWithMediaType:AVMediaTypeVideo]
firstObject];
[videoTrack insertTimeRange:videoTimeRange
ofTrack:assetTrack
atTime:cursorTime error:nil];

// Increment cursor time
cursorTime = CMTimeAdd(cursorTime,

```

```

videoDuration); // 4

// Extract and insert Tea Garden segment
assetTrack
=
5
[[teaGardenAsset tracksWithMediaType:AVMediaTypeVideo]
firstObject];
[videoTrack insertTimeRange:videoTimeRange
ofTrack:assetTrack
atTime:cursorTime error:nil];

// Reset cursor time
cursorTime =
kCMTimeZero; // 6
CMTIME audioDuration = composition.duration;
CMTimeRange audioTimeRange = CMTimeRangeMake(kCMTimeZero,
audioDuration);

// Extract and insert Tea Garden segment
assetTrack
=
7
[[soundtrackAsset tracksWithMediaType:AVMediaTypeAudio]
firstObject];
[audioTrack insertTimeRange:audioTimeRange
ofTrack:assetTrack
atTime:cursorTime error:nil];

```

- 1.** You define a `CMTIME` defining what I refer to as the insertion cursor. This is the point along the track where you insert the media segment.
- 2.** The goal is to capture the first 5 seconds of each video, so you create a `CMTimeRange` starting at `kCMTimeZero` with a duration time of 5 seconds.
- 3.** Use `tracksWithMediaType:` method to extract the video track from first AVAsset. This method returns an array of tracks matching the given media type, but because this media contains only a single video track, you'll grab the first object. You then call the `insertTimeRange:ofTrack:atTime:error` method on the video track to insert the video segment into it.
- 4.** Move the cursor's insertion time by calling the `CMTimeAdd` function, adding the `videoDuration` to the current `cursorTime` value. This

moves the cursor time ahead so the next media segment will be cleanly inserted at the end of the other.

5. Just as you did in step 3, you extract the asset's video track and insert it into the composition's video track.
6. You'd like the audio track to span the full duration of the video clips, so you begin by resetting the `cursorTime` back to `kCMTimeZero`. You then find the total duration of the composition and create a `CMTimeRange` that spans the whole composition time.
7. Extract the audio track from the audio asset and insert it into the composition's audio track.

You now have a composition that looks like the one shown in [Figure 9.6](#).

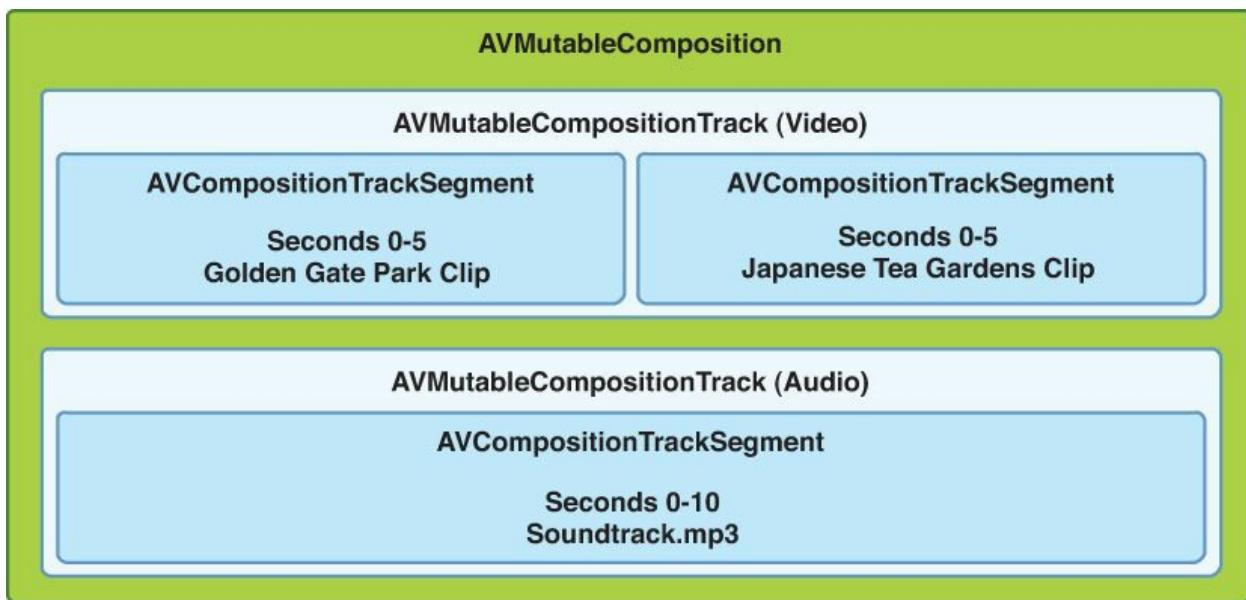


Figure 9.6 Completed composition

This composition could now be used like any other `AVAsset`, which means it can be played, exported, or processed. The capability of building multitrack compositions from a variety of media sources is a very powerful and useful feature provided by AV Foundation.

Introducing 15 Seconds

The application you'll be building over the course of the next several chapters is a video editing application called *15 Seconds* (see [Figure 9.7](#)). You'll find a starter project in the Chapter 9 directory called **FifteenSeconds_Starter**.

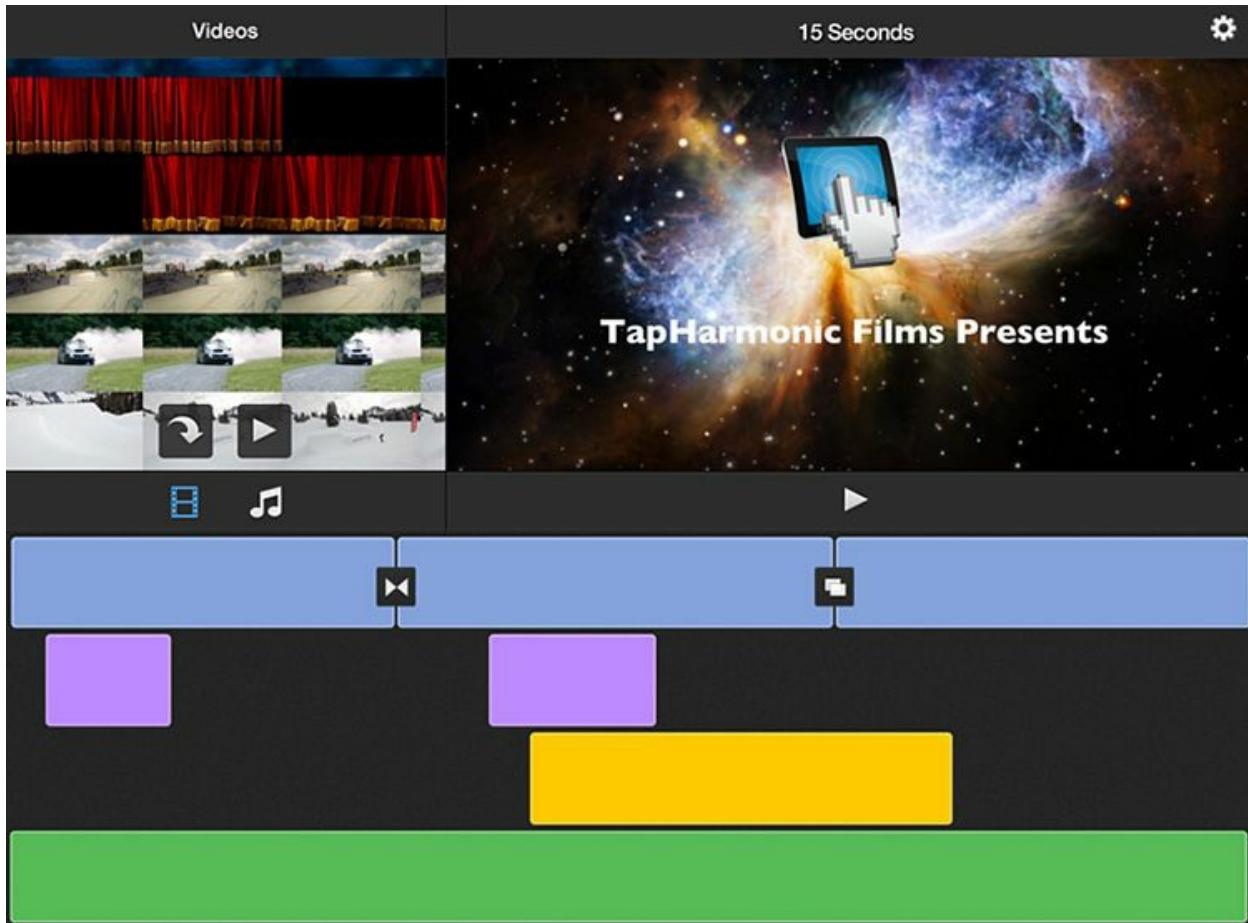


Figure 9.7 15 Seconds interface

As you can probably infer from the name, this application enables you to build 15-second media compositions. It incorporates many of the things you've done up to this point, including video playback, reading metadata, and image extraction, and provides the foundation for learning the capabilities of AV Foundation editing. Let's discuss some of the essentials of this application to get a better understanding for how it works.

View Controller Composition

There are three distinct sections of this application: a media picker for selecting and previewing audio and video clips, a video player, and a timeline area enabling you to arrange the media clips in your composition. [Figure 9.8](#) provides an illustration of the application's view controller objects.



Figure 9.8 15 Seconds view controller composition

THMainViewController is a container view controller that manages its child view controllers and acts as the central mediator for request handling in the application. For instance, when a request to preview a media item is received from either the THVideoPickerController or THAudioPickerController, the main view controller is passed the selected media item, creates an AVPlayerItem from it, and passes it for playback to the THPlayerViewController. Likewise, when the user taps the player's Play button, the main view controller receives the request, asks the THTimelineViewController for its current state, and then builds a playable form of the timeline to pass back to the player view controller. We won't go into the details of the responsibilities of each view controller, because they are either performing tasks that were covered earlier in the book or are not relevant to the topic at hand. However, I would encourage you to look through the application's source to better understand how the pieces fit together.

The part of the application that you will be building is the *model*, which requires some understanding of the application's data model. [Figure 9.9](#) provides a class diagram describing the core objects composing the data model for the 15 Seconds app.

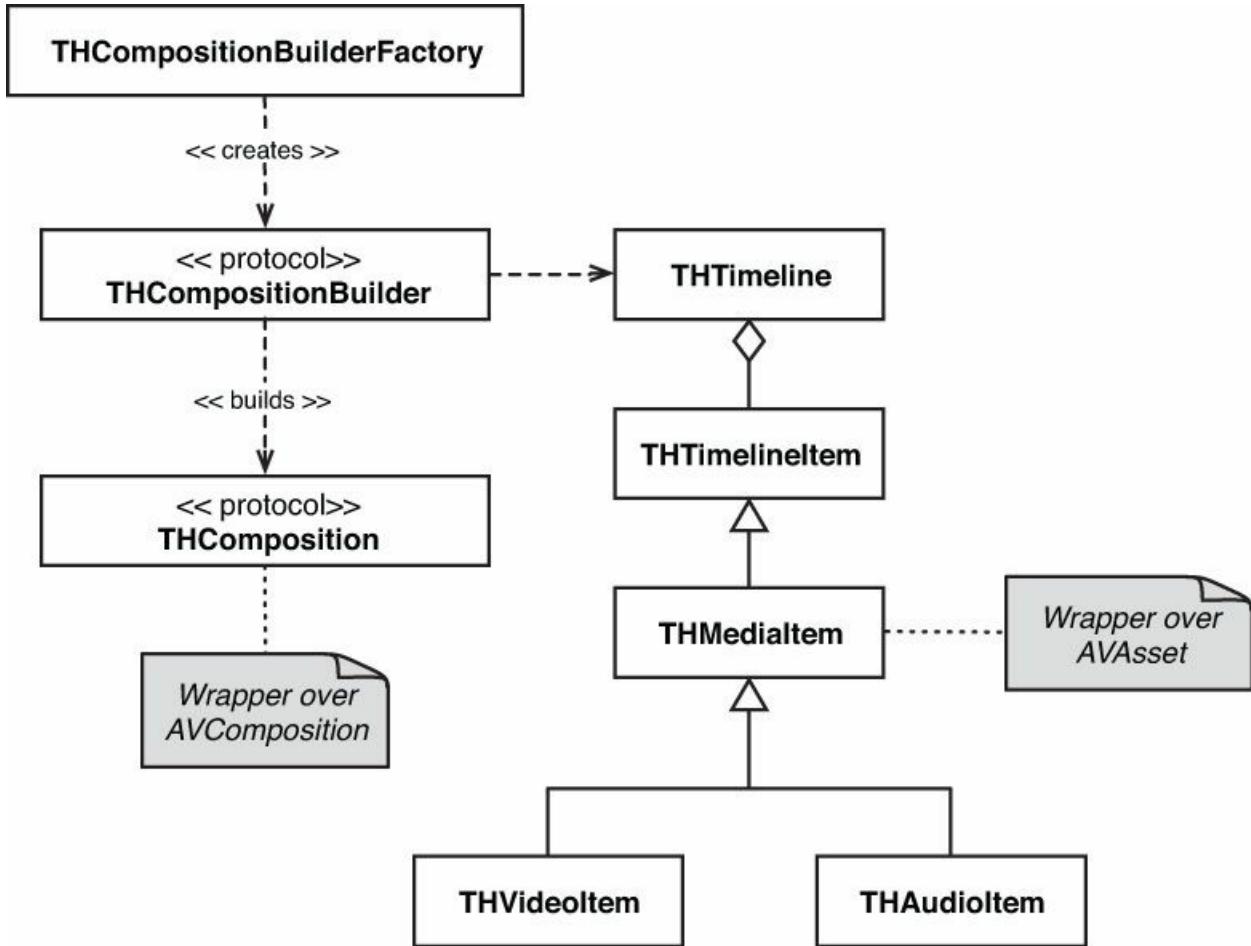


Figure 9.9 15 Seconds object model

An object called **THTimeline** backs the application's timeline area. This object is composed of one or more instances of **THTimelineItem**, which are represented visually as the colored boxes you see in the user interface. A **THTimelineItem** has no dependency on AV Foundation but defines the item's duration and its location along the timeline. When working with audio and video media, you'll use a subclass of the timeline item called **THMediaItem**. This class acts as a wrapper over an instance of **AVAsset** and is responsible for properly loading and preparing the asset for use in a composition. **THMediaItem** is further subclassed by **THVideoItem** and **THAudioItem** for working with video and audio assets, respectively. In this

chapter, the distinction between an audio item and video item is not particularly important, but will take on greater relevance in subsequent chapters.

When a request is made to play the current state of the timeline, the `THMainViewController` collaborates with a number of objects to handle the request. It begins by getting the current `THTimeline` instance from the timeline view controller and retrieves an instance of `THCompositionBuilder` from an associated factory object. `THCompositionBuilder` is a protocol defining the interface to build instances of `THComposition`. A concrete composition builder is responsible for building the actual `AVComposition` and its associated tracks and segments and wrapping it in an instance of `THComposition`. This is another protocol type acting as a wrapper over the `AVComposition` instance and providing methods to convert the composition to either a *playable* or *exportable* form. The core AV Foundation work happens in the `THCompositionBuilder` and `THComposition` objects, so you'll be building concrete implementations of these classes in this chapter and the upcoming chapters as well.

Now that you have a better understanding of how that application and its associated data model objects are composed, it's time to begin building out the application's functionality.

Building a Composition

The first task is to build an object adopting the `THComposition` protocol (see [Listing 9.1](#)).

Listing 9.1 **THComposition Protocol**

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>

@protocol THComposition <NSObject>

- (AVPlayerItem *)makePlayable;
- (AVAssetExportSession *)makeExportable;

@end
```

This provides a common interface for the alternate implementations you'll be building over the next few chapters. It provides the interface to create a playable version of the composition as well as an exportable version, which will be used by the `THMainViewController` class. Your first implementation of this protocol will be a class called `THBasicComposition` (see [Listing 9.2](#)).

Listing 9.2 **THBasicComposition** Interface

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>
#import "THComposition.h"

@interface THBasicComposition : NSObject <THComposition>

@property (strong, readonly, nonatomic) AVComposition
*composition;

+ (instancetype)compositionWithComposition:(AVComposition *)
composition;
- (instancetype)initWithComposition:(AVComposition *)composition;

@end
```

This is a simple interface with two variant initializers and a read-only pointer to the underlying `AVComposition` object. Let's build out the implementation of this class in [Listing 9.3](#).

Listing 9.3 **THBasicComposition** Implementation

[Click here to view code image](#)

```
#import "THBasicComposition.h"

@interface THBasicComposition ()
@property (strong, nonatomic) AVComposition *composition;
@end

@implementation THBasicComposition

+ (id)compositionWithComposition:(AVComposition *)composition {
    return [[self alloc] initWithComposition:composition];
}

- (id)initWithComposition:(AVComposition *)composition {
```

```

        self = [super init];
        if (self) {
            _composition = composition;
        }
        return self;
    }

- (AVPlayerItem *)makePlayable
{
    // 1
    return [AVPlayerItem playerItemWithAsset:[self.composition
copy]];
}

- (AVAssetExportSession *)makeExportable
{
    // 2
    NSString *preset = AVAssetExportPresetHighestQuality;
    return [AVAssetExportSession exportSessionWithAsset:
[self.composition copy]
                                presetName:preset];
}

@end

```

1. The `makePlayable` method will create an `AVPlayerItem` from the underlying `AVComposition` instance. `AVComposition` adopts the `NSCopying` protocol, so a good practice to follow is to make an immutable copy of this object by calling its `copy` method. This guards against the mutable state of this object changing while being presented.
2. In the `makeExportable` method you create a new instance of `AVAssetExportSession`, again making a copy of the `AVMutableComposition` to pass to the `exportSessionWithAsset:presetName:` initializer.

You have a completed implementation of the `THComposition` protocol. Now it's time to create a builder implementation that will construct an instance of this object.

Building the Composition Builder

For every `THComposition` implementation, the application also has an associated `THCompositionBuilder` responsible for building an instance of it. This protocol is shown in [Listing 9.4](#).

Listing 9.4 THCompositionBuilder Protocol

[Click here to view code image](#)

```
#import <AVFoundation/AVFoundation.h>
#import "THComposition.h"

@protocol THCompositionBuilder <NSObject>
- (id <THComposition>)buildComposition;
@end
```

A concrete instance of this protocol is responsible for providing an implementation of the `buildComposition` method, which creates the `AVComposition` and its associated tracks and track segments. You'll create an object called `THBasicCompositionBuilder` that adopts this protocol. [Listing 9.5](#) provides the interface for this class.

Listing 9.5 THBasicCompositionBuilder Interface

[Click here to view code image](#)

```
#import "THCompositionBuilder.h"
#import "THTimeline.h"

@interface THBasicCompositionBuilder : NSObject
<THCompositionBuilder>
- (id)initWithTimeline:(THTimeline *)timeline;
@end
```

This class adopts the `THCompositionBuilder` protocol and provides an `initWithTimeline :` method, which is passed an instance of `THTimeline`. The timeline object holds references to all the `THTimelineItem` instances and their underlying instances of `AVAsset`, which will provide you with everything needed to build an `AVComposition`. Let's move on to the implementation of this class in [Listing 9.6](#).

Listing 9.6 THBasicCompositionBuilder Implementation

[Click here to view code image](#)

```

#import "THBasicCompositionBuilder.h"
#import "THBasicComposition.h"
#import "THFunctions.h"

@interface THBasicCompositionBuilder ()
@property (strong, nonatomic) THTimeline *timeline;
@property (strong, nonatomic) AVMutableComposition *composition;
@end

@implementation THBasicCompositionBuilder

- (id)initWithTimeline:(THTimeline *)timeline {
    self = [super init];
    if (self) {
        _timeline = timeline;
    }
    return self;
}

- (id <THComposition>)buildComposition {

    self.composition = [AVMutableComposition
composition]; // 1

    [self addCompositionTrackOfType:AVMediaTypeVideo
withMediaItems:self.timeline.videos];

    [self addCompositionTrackOfType:AVMediaTypeAudio
withMediaItems:self.timeline.voiceOvers];

    [self addCompositionTrackOfType:AVMediaTypeAudio
withMediaItems:self.timeline.musicItems];

    // Create and return the basic
composition // 2
    return [THBasicComposition
compositionWithComposition:self.composition];
}

- (void)addCompositionTrackOfType:(NSString *)mediaType
withMediaItems:(NSArray *)mediaItems {

    // To be implemented
}

@end

```

1. In the buildComposition method you begin by creating a new

instance of `AVMutableComposition`. You'll call the private `addCompositionTrackOfType:with-MediaItems` method for each of the elements contained in the timeline. You'll provide an implementation of this method shortly.

2. Finally, when the composition and all its tracks have been built, you create an instance of the `THBasicComposition` object you created earlier, passing it a reference to the newly created `AVMutableComposition` object.

Let's move on to the implementation of the private `addCompositionTrackOfType:with-MediaItems` method, which is doing the heavy lifting in this class (see [Listing 9.7](#)).

Listing 9.7 Building the Track Contents

[Click here to view code image](#)

```
@implementation THBasicCompositionBuilder

...
- (void)addCompositionTrackOfType:(NSString *)mediaType
    withMediaItems:(NSArray *)mediaItems {
    if (!THIsEmpty(mediaItems)) // 1
    {
        CMPersistentTrackID trackID =
        kCMPersistentTrackID_Invalid;
        AVMutableCompositionTrack *compositionTrack
        = // 2
            [self.composition
        addMutableTrackWithMediaType:mediaType
                           preferredTrackID:trackID];
        // Set insert cursor to 0
        CMTime cursorTime =
        kCMTimeZero; // 3
        for (THMediaItem *item in mediaItems) {
            if
            (CMTIME_COMPARE_INLINE(item.startTimeInTimeline, // 4
                                  !=
                                  kCMTimeInvalid)) {
                cursorTime = item.startTimeInTimeline;
```

```

        }

        AVAssetTrack *assetTrack
        =
            [[item.asset tracksWithMediaType:mediaType]
firstObject];

            [compositionTrack
insertTimeRange:item.timeRange
                                // 6
ofTrack:assetTrack
atTime:cursorTime
error:nil];

        // Move cursor to next item time
        cursorTime = CMTimeAdd(cursorTime,
item.timeRange.duration);    // 7
    }
}

@end

```

- 1.** Using the `THIsEmpty` function, you begin by validating that the `mediaItems` array passed to this method is not empty. This is a generic function used to validate a variety of the Foundation framework's types to ensure it contains valid content.
- 2.** For each invocation of this method, you create a new instance of `AVMutableCompositionTrack` with the specified media type. Use the constant `kCMPersistentTrackID_Invalid` to indicate AV Foundation should automatically generate an appropriate track ID. The result of this will cause the track IDs to be given a `1..n` track ID assignment.
- 3.** Create a new `CMTIME` with the constant `kCMTIMEZero`. This will be used as the insert cursor time.
- 4.** Perform a quick check to determine that the timeline item's `startTimeInTimeline` returns a valid `CMTIME` value. The clips contained in the timeline's video and music tracks will return `kCMTIMEInvalid` for their `startTimeInTimeline` because they are always laid out sequentially, meaning the application doesn't allow for gaps between clips. However, the voice-over track allows you to slide the clip to any arbitrary point along the timeline. In this case

you'll want to modify the `cursorTime` to account for the user-defined position of this clip in the timeline.

5. From the item's underlying `AVAsset` instance, you extract the track corresponding to the requested `mediaType`, which will be either `AVMediaTypeVideo` or `AVMediaTypeAudio`. This method returns an `NSArray` of tracks matching the specified media type, but all media contained in the demo project will contain only a single track of a given type, so you'll ask for the first object in the array.
6. Insert the media clip into the composition track using the item's specified `timeRange` at the computed `cursorTime` value.
7. Finally, compute a new `cursorTime` value by adding the current clip's duration to the `cursorTime` so it will be correctly set for the next iteration of the loop.

Now you've come to the moment of truth. Run the application and add three video clips from the video picker to the timeline, add a single music track, and a single voice-over track and press the Play button to see and hear your handiwork in action. You can drag the voice-over clip to a new location along the timeline and drag and drop the video clips into an alternate order and press Play again. You have just built a real video editor!

Exporting the Composition

Having the capability of building a composition is certainly a lot fun; however, it would be for nothing if you can't share your creativity with the world. You need to be able to export this composition, so let's build out this functionality. If you tap the gear icon on the upper-right corner of the screen, you'll see it contains an item labeled Export Composition. Currently, tapping this item doesn't perform the desired functionality, so let's build out this functionality in an object called `THCompositionExporter` (see [Listing 9.8](#)).

Listing 9.8 `THCompositionExporter` Interface

[Click here to view code image](#)

```
#import "THTimeline.h"
#import "THComposition.h"

@interface THCompositionExporter : NSObject
```

```
@property (nonatomic) BOOL exporting;
@property (nonatomic) CGFloat progress;

- (instancetype)initWithComposition:(id
<THComposition>)composition;

- (void)beginExport;

@end
```

The key method this object defines is `beginExport`, which will perform the actual export process. The state of the `exporting` and `progress` properties will be observed by `THMainViewController` to update the state of the user interface while the export is in progress. Let's move on and begin implementing this class in [Listing 9.9](#).

Listing 9.9 Exporting a Composition

[Click here to view code image](#)

```
#import "THCompositionExporter.h"
#import "UIAlertView+THAdditions.h"
#import <AssetsLibrary/AssetsLibrary.h>

@interface THCompositionExporter ()
@property (strong, nonatomic) id <THComposition> composition;
@property (strong, nonatomic) AVAssetExportSession *exportSession;
@end

@implementation THCompositionExporter

- (instancetype)initWithComposition:(id
<THComposition>)composition {

    self = [super init];
    if (self) {
        _composition = composition;
    }
    return self;
}

- (void)beginExport {

    self.exportSession = [self.composition
makeExportable]; // 1
    self.exportSession.outputURL = [self exportURL];
    self.exportSession.outputFileType = AVFileTypeMPEG4;
```

```

    [self.exportSession
exportAsynchronouslyWithCompletionHandler:^{           // 2
    // To be implemented
}];

    self.exporting =
YES;                                         // 3
    [self
monitorExportProgress];
4
}

- (void)monitorExportProgress {
    // To be implemented
}

- (NSURL *)exportURL {
    NSString *filePath = nil;
    NSUInteger count = 0;
    do {
        filePath = NSTemporaryDirectory();
        NSString *numberString = count > 0 ?
            [NSString stringWithFormat:@"%li", (unsigned long)
count] : @"";
        NSString *fileNameString =
            [NSString stringWithFormat:@"Masterpiece-%@.m4v",
numberString];
        filePath = [filePath
stringByAppendingPathComponent:fileNameString];
        count++;
    } while ([[NSFileManager defaultManager]
fileExistsAtPath:filePath]);

    return [NSURL fileURLWithPath:filePath];
}

@end

```

- 1.** Begin by creating an exportable version of this composition. This returns an instance of `AVAssetExportSession` properly configured with the underlying `AVComposition` instance and an appropriate session preset. You dynamically generate a unique output URL for the session as well as set the output file type.
- 2.** Call the `exportAsynchronouslyWithCompletionHandler:`

method to begin the export process. You temporarily supply an empty handler block as the method's argument.

3. Set the `exporting` property to YES.

`THMasterViewController` will observe this change and will present the progress user interface.

4. Finally, you call the `monitorExportProgress` method, which will poll the state of the export session to determine its current progress.

Let's move on and provide an implementation of the `monitorExportProgress` method in [Listing 9.10](#).

Listing 9.10 Monitoring the Export Progress

[Click here to view code image](#)

```
@implementation THCompositionExporter

...
- (void)monitorExportProgress {
    double delayInSeconds = 0.1;
    int64_t delta = (int64_t)delayInSeconds * NSEC_PER_SEC;
    dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW,
delta);

    dispatch_after(popTime, dispatch_get_main_queue(),
^{
    AVAssetExportSessionStatus status =
self.exportSession.status;

    if (status == AVAssetExportSessionStatusExporting)
    {
        self.progress = self.exportSession.progress;
        [self monitorExportProgress];

        } else {
            self.exporting = NO;
        }
    });
}

...
@end
```

1. Begin by enqueueing a block to be executed after a short delay that will check the status of the export process.

2. Check the export session's `status` property to determine its current state. If the status returns

`AVAssetExportSessionStatusExporting`, you update the `progress` property with the current value of the export session's `progress` and recursively call the `monitorExportProgress` method. If the export session's `status` property returns any other status value, you set the `exporting` property to `NO` so the user interface can be updated appropriately.

The final bit of functionality to implement is to handle the export completion and write the exported file to the Assets Library (see [Listing 9.11](#)).

Listing 9.11 Handling the Export Completion

[Click here to view code image](#)

```
@implementation THCompositionExporter

...
- (void)beginExport {
    self.exportSession = [self.composition makeExportable];
    self.exportSession.outputURL = [self exportURL];
    self.exportSession.outputFileType = AVFileTypeMPEG4;

    [self.exportSession
    exportAsynchronouslyWithCompletionHandler:^{
        dispatch_async(dispatch_get_main_queue(),
        ^{
            AVAssetExportSessionStatus status =
self.exportSession.status;
            if (status == AVAssetExportSessionStatusCompleted) {
                [self writeExportedVideoToAssetsLibrary];
            } else {
                UIAlertView *alertView =
UIAlertView *alertView initWithTitle:@"Export Failed"
                                         message:@"The requested
export failed."];
            }
        });
    }];
}

self.exporting = YES;
```

```

        [self monitorExportProgress];
    }

- (void)writeExportedVideoToAssetsLibrary {
    NSURL *exportURL = self.exportSession.outputURL;
    ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];

    if ([library
videoAtPathIsCompatibleWithSavedPhotosAlbum:exportURL]) { // 2

        [library
writeVideoAtPathToSavedPhotosAlbum:exportURL           // 3
completionBlock:^(NSURL
*assetURL,
NSError
*error) {

            if (error)
                // 4
            NSString *message = @"Unable to write to Photos
library.";
            UIAlertView showAlertWithTitle:@"Write Failed"
message:message];
        }

        [[NSFileManager defaultManager]
removeItemAtURL:exportURL
error:nil];
    ];
} else {
    NSLog(@"Video could not be exported to the assets
library.");
}
}

@end

```

- 1.** In the handler block, begin by dispatching back to the main queue and checking the export session status. If the export completed successfully, you write the exported video to the Assets Library. If the export failed, you show an error alert to the user.
- 2.** Create a new instance of `ALAssetsLibrary`. Before attempting to write to the library, it is a good practice to validate that what you are attempting to write can be written by calling the `videoAtPathIsCompatibleWithSavedPhotosAlbum:` method.

3. Call

```
writeVideoAtPathToSavedPhotosAlbum:completionBlock:
```

to begin the write operation. The first time this is attempted, you'll receive a security prompt asking for your permission to write to the Photos Library. Be sure to answer yes.

4. Finally, in the completion handler, determine if any errors were encountered. If so, show an appropriate error alert to the user. Regardless of the outcome of the write operation, you'll want to delete the exported file from the user's temporary directory.

Run the application. You can either manually build a new composition or select Load Default Composition from the menu. Tap the gear icon in the corner of the screen and select the Export Composition menu item. You'll now see a progress view over the video player indicating the progress of the export. When the export completes, you can switch over to the Photos app and see your masterpiece in action.

Summary

In this chapter you got your first introduction to AV Foundation's media-editing capabilities. You used `AVComposition` to build a multitrack project, composing several video and audio assets to create a new and unique piece of media. You also saw how compositions could be used like any other `AVAsset` for tasks such as playback and export. Although our discussion focused on building a traditional video-editing application, you'll find that `AVComposition` can be useful in a variety of situations.

You're off to a great start, but the fun is just beginning! Over the next three chapters we'll discuss how you can take your basic *cuts-only* editing application and add a number of advanced features that will further extend the features and capabilities of the app.

Challenge

Learning to use `CMTime` and `CMTimeRange` is one of the most critical things to learn to effectively use AV Foundation's editing facilities. Thinking of time as a rational number is a common stumbling block when you're learning to use the editing APIs. Inside the [Chapter 9](#) directory, you'll find a simple Mac command-line project called **TimeChallenge** that you can use to learn and experiment with these data types and functions. The project

provides several examples and suggested exercises for you to try.

10. Mixing Audio

The 15 Seconds app is coming along nicely. It provides the capability of building multitrack, cuts-only compositions that can be played in the application and exported to share with the world. The core functionality is working well, and now it's time to turn our attention to some additional features provided by AV Foundation to enhance and refine the compositions the application creates. In this chapter we'll discuss how to improve the audio aspects of your compositions by performing some basic audio mixing. This will be a fairly short chapter, but the lessons learned will help you make big improvements to the quality of your audio output.

Mixing Audio

The compositions you created in the previous chapter looked good, providing nice clean cuts between scenes, but there were definitely a couple issues to resolve relating to the audio output. The first is that the music track immediately comes in at full volume when playback begins and then abruptly cuts off at the end of the composition, which results in a fairly jarring experience for the listener. It would be better if you could gently fade the music in at the beginning of the track and then fade it back out at the end. The other issue, and arguably the more serious concern, has to do with the voiceover track. The volume of the music track overwhelms the voiceover, making it very difficult to hear. Instead of letting both audio tracks fight for the listener's attention, it would be better to apply a technique called *ducking* where you quickly ramp the volume down while the voiceover plays, hold it constant for the duration of the voiceover, and ramp it back to its previous volume when it's done. A class provided by the framework called `AVAudioMix` can help you address both concerns.

An `AVAudioMix` is used to apply custom audio processing to your composition's audio tracks. [Figure 10.1](#) shows `AVAudioMix` and its related classes, as well as the classes in the framework that make use of it.

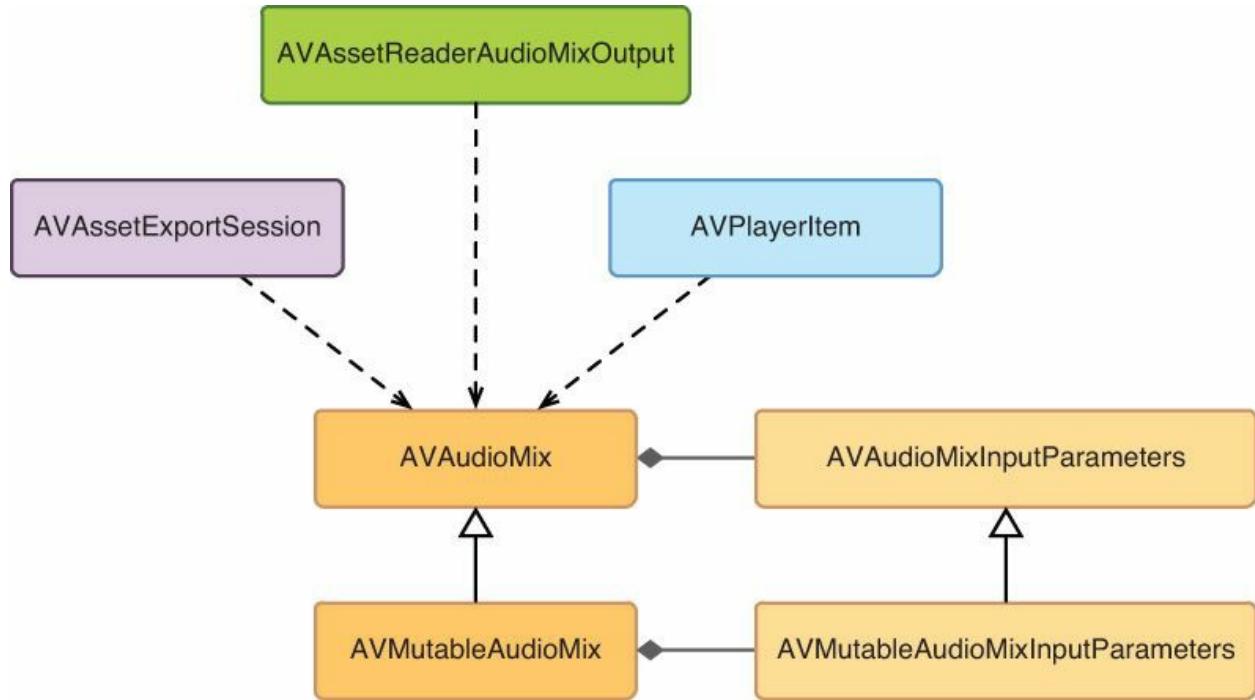


Figure 10.1 AVAudioMix classes

The nature of the audio processing to be performed by an `AVAudioMix` is defined by its collection of input parameters, which are provided in the form of an object called `AVAudioMixInputParameters`. An instance of `AVAudioMixInputParameters` is associated with a single audio track in the composition and defines the audio processing to be performed on the track when added to the audio mix. `AVAudioMix` and its associated collection of `AVAudioMixInputParameters` are immutable objects, which means they're suitable for providing the relevant data to clients such as `AVPlayerItem` and `AVAssetExportSession`, but they don't provide the means to manipulate their state. When you need to create a custom audio mix, you'll instead turn to their mutable subclasses found in `AVMutableAudioMix` and `AVMutableAudioMixInputParameters`.

Automating Volume Changes

The most essential audio processing that can be applied via an audio mix is to adjust the volume levels of your composition's audio tracks. When a composition is played or exported, the default behavior is for the audio tracks to play at their full or natural volume. This may be acceptable in a composition with a single audio track, but becomes problematic when your

compositions contain multiple audio sources. With multiple audio tracks, the volume of each competes for space, which inevitably results in some audio that may be barely audible. Some simple audio mixing can resolve this problem and help you achieve a mix that's crisp and clear.

If you've performed any audio mixing in tools such as Pro Tools or Logic Pro X, you're used to thinking of volume in terms of decibels. AV Foundation uses a simpler model where volume is defined as a floating-point value on a normalized scale from 0.0 to 1.0, with 0.0 equaling silence and 1.0 equaling full volume. The default volume of an audio track is set to 1.0, but can be modified using an instance of `AVMutableAudioMixInputParameters`. This object enables you to define automated volume changes to be made at a specific point in time or over a given time range (see [Figure 10.2](#)).

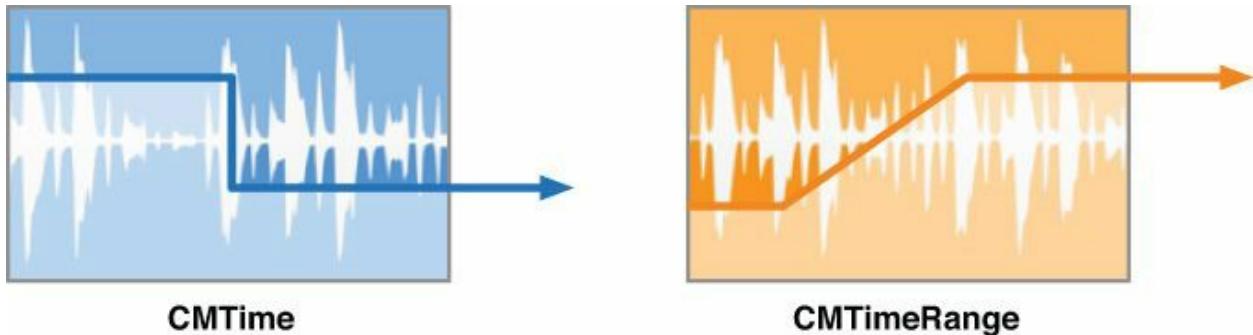


Figure 10.2 Time-based volume changes

`AVMutableAudioMixInputParameters` provides two methods to achieve the volume changes depicted in [Figure 10.2](#):

- `setVolume:atTime:`: enables you to make an immediate volume adjustment at the specified time. The volume will hold constant at the volume level specified for the duration of the audio track or until another volume adjustment is encountered.
- `setVolumeRampFromStartVolume:toEndVolume:timeRange:`: enables you to smoothly ramp the volume from one value to another over a given time range. When the time range to perform this volume adjustment is encountered, the volume immediately jumps to the specified starting volume and gradually adjusts to the ending volume over the duration of the time range. The volume is held constant at the value specified in the `toEndVolume:` argument for the duration of the audio track or until another volume adjustment is reached.

Let's take a look at an example of how to use these methods to create the volume automation shown in [Figure 10.3](#).

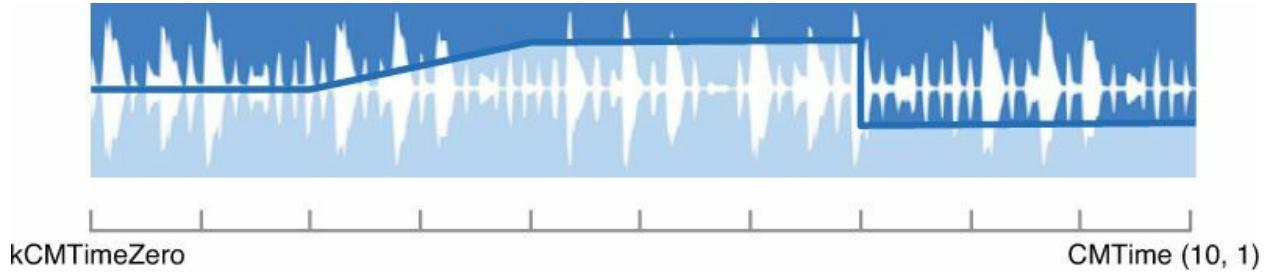


Figure 10.3 Example volume automation

[Click here to view code image](#)

```
AVCompositionTrack *track = // audio track in composition

// Define automation times
CMTime twoSeconds = CMTimeMake(2, 1);
CMTime fourSeconds = CMTimeMake(4, 1);
CMTime sevenSeconds = CMTimeMake(7, 1);

// Create a new parameters object for the given audio track
AVMutableAudioMixInputParameters *parameters =
    [AVMutableAudioMixInputParameters
audioMixInputParametersWithTrack:track];

// Set initial volume
[parameters setVolume:0.5f atTime:kCMTTimeZero];

// Define time range of the volume ramp
CMTimeRange range = CMTimeRangeFromTimeToTime(twoSeconds,
fourSeconds);

// Perform 2 second ramp from 0.5 -> 0.8
[parameters setVolumeRampFromStartVolume:0.5f toEndVolume:0.8f
timeRange:range];

// Drop volume to 0.3 at the 7-second mark
[parameters setVolume:0.3f atTime:sevenSeconds];

// Create a new audio mix instance
AVMutableAudioMix *audioMix = [AVMutableAudioMix audioMix];

// Assign the input parameters to the audio mix
audioMix.inputParameters = @*[parameters];
```

The example begins by creating a new instance of `AVMutableAudioMixInputParameters` associated with the track to

be operated on. A track's default volume is 1.0, so to provide a starting volume of 0.5 as depicted in [Figure 10.3](#), it sets the initial volume using the `setVolume:atTime:` method passing a time value of `kCMTimeZero`. At 2 seconds into the track, it applies a volume ramp from 0.5 to 0.8 over a 2-second duration using the `setVolumeRampFromStartVolume:toEndVolume:timeRange:` method. Finally, at the 7-second mark, it drops the volume to 0.3 using the `setVolume:atTime:` method. With all parameters defined it creates an `AVMutableAudioMix`, adds the parameters to an array, and assigns it as the audio mix's `inputParameters` property. The example builds a fully formed audio mix that can now be set as the `audioMix` property on an `AVPlayerItem` or `AVAssetExportSession` for playback or export.

Note

We're discussing `AVAudioMix` and `AVAudioMixInputParameters` in the context of mixing the audio in an `AVComposition`. Although this is likely the most common use case, it's not solely for use with a composition. An instance of `AVAudioMixInputParameters` is associated with an `AVAssetTrack` (`AVCompositionTrack`'s superclass), which means you can define volume adjustments on the audio tracks in a normal `AVAsset`, which could be useful in certain playback and export scenarios.

Mixing Audio in the 15 Seconds App

Now that you have an understanding of `AVAudioMix`, let's put this into action in the 15 Seconds app. Begin by creating a new class called `THAudioMixComposition` that adopts the `THComposition` protocol, as shown in [Listing 10.1](#).

Listing 10.1 **THAudioMixComposition** Interface

[Click here to view code image](#)

```
#import "THComposition.h"

@interface THAudioMixComposition : NSObject <THComposition>
```

```

@property (strong, nonatomic, readonly) AVAudioMix *audioMix;
@property (strong, nonatomic, readonly) AVComposition
*composition;

+ (instancetype)compositionWithComposition:(AVComposition
*)composition
                                    audioMix:(AVAudioMix *)audioMix;

- (instancetype)initWithComposition:(AVComposition *)composition
                                audioMix:(AVAudioMix *)audioMix;

@end

```

This interface should look familiar because it's very similar to the THBasicComposition object you created in the previous chapter. The key difference is the addition of the AVAudioMix property and the initializers that take both an AVComposition and an AVAudioMix argument. Let's move on to the implementation in [Listing 10.2](#).

Listing 10.2 **THAudioMixComposition** Implementation

[Click here to view code image](#)

```

#import "THAudioMixComposition.h"

@interface THAudioMixComposition ()
@property (strong, nonatomic) AVAudioMix *audioMix;
@property (strong, nonatomic) AVComposition *composition;
@end

@implementation THAudioMixComposition

+ (instancetype)compositionWithComposition:(AVComposition
*)composition
                                    audioMix:(AVAudioMix *)audioMix
{
    return [[self alloc] initWithComposition:composition
audioMix:audioMix];
}

- (instancetype)initWithComposition:(AVComposition *)composition
                                audioMix:(AVAudioMix *)audioMix {
    self = [super init];
    if (self) {
        _composition = composition;
        _audioMix = audioMix;
    }
}

```

```

        return self;
    }

- (AVPlayerItem *)makePlayable
{
    // 1
    AVPlayerItem *playerItem =
    [AVPlayerItem playerItemWithAsset:[self.composition
copy]];
    playerItem.audioMix = self.audioMix;
    return playerItem;
}

- (AVAssetExportSession *)makeExportable
{
    // 2
    NSString *preset = AVAssetExportPresetHighestQuality;
    AVAssetExportSession *session =
    [AVAssetExportSession exportSessionWithAsset:
    [self.composition copy]
    presetName:preset];
    session.audioMix = self.audioMix;
    return session;
}

@end

```

1. In the `makePlayable` method you create an `AVPlayerItem` with the `AVComposition` instance. You set the `audioMix` object as the player item's `audioMix` property. This enables the audio processing to be applied when played in the application's video player.
2. In the `makeExportable` method you create an `AVAssetExportSession` as you did in the previous chapter. You set the `audioMix` object as the export session's `audioMix` property enabling the audio processing to be applied when exporting this composition.

`THAudioMixComposition` is complete, so let's move on and discuss implementing its associated `THCompositionBuilder` object. Begin by creating a new object called `THAudioMixCompositionBuilder` adopting the `THCompositionBuilder` protocol, as shown in [Listing 10.3](#).

Listing 10.3 **THAudioMixCompositionBuilder** Interface

[Click here to view code image](#)

```

#import "THCompositionBuilder.h"
#import "THTimeline.h"

@interface THAudioMixCompositionBuilder : NSObject
<THCompositionBuilder>

- (id)initWithTimeline:(THTimeline *)timeline;

@end

```

The `THAudioMixCompositionBuilder` has a simple interface that accepts the `THTimeline` instance. Recall that the `THTimeline` object contains the state of the application's timeline area and provides the data necessary to build the composition object. Let's begin the implementation of this class as shown in [Listing 10.4](#).

Listing 10.4 `THAudioMixCompositionBuilder` Implementation

[Click here to view code image](#)

```

#import "THAudioMixCompositionBuilder.h"
#import "THAudioItem.h"
#import "THVolumeAutomation.h"
#import "THAudioMixComposition.h"
#import "THFunctions.h"

@interface THAudioMixCompositionBuilder ()
@property (strong, nonatomic) THTimeline *timeline;
@property (strong, nonatomic) AVMutableComposition *composition;
@end

@implementation THAudioMixCompositionBuilder

- (id)initWithTimeline:(THTimeline *)timeline {
    self = [super init];
    if (self) {
        _timeline = timeline;
    }
    return self;
}

- (id <THComposition>)buildComposition {

    self.composition = [AVMutableComposition
composition]; // 1

    [self addCompositionTrackOfType:AVMediaTypeVideo
withMediaItems:self.timeline.videos];
}

```

```

    [self addCompositionTrackOfType:AVMediaTypeAudio
        withMediaItems:self.timeline.voiceOvers];

    AVMutableCompositionTrack *musicTrack =
        [self addCompositionTrackOfType:AVMediaTypeAudio
            withMediaItems:self.timeline.musicItems];

    AVAudioMix *audioMix = [self
buildAudioMixWithTrack:musicTrack];           // 2

    return [THAudioMixComposition
compositionWithComposition:self.composition
                                audioMix:audioMix];
}

- (AVMutableCompositionTrack *)addCompositionTrackOfType:(NSString
*)type
                                withMediaItems:(NSArray
*)mediaItems {

    if (!THIsEmpty(mediaItems)) {

        CMPersistentTrackID trackID =
kCMPersistentTrackID_Invalid;

        AVMutableCompositionTrack *compositionTrack =
            [self.composition addMutableTrackWithMediaType:type
                                preferredTrackID:trackID];
        // Set insert cursor to 0
        CMTime cursorTime = kCMTimeZero;

        for (THMediaItem *item in mediaItems) {

            if (CMTIME_COMPARE_INLINE(item.startTimeInTimeline,
                                      !=,
                                      kCMTimeInvalid)) {
                cursorTime = item.startTimeInTimeline;
            }

            AVAssetTrack *assetTrack =
                [[item.asset tracksWithMediaType:type]
firstObject];

            [compositionTrack insertTimeRange:item.timeRange
                                ofTrack:assetTrack
                                atTime:cursorTime
                                error:nil];

            // Move cursor to next item time
        }
    }
}

```

```

        cursorTime = CMTimeAdd(cursorTime,
item.timeRange.duration);
    }

    return compositionTrack;
}

return nil;
}

- (AVAudioMix *)buildAudioMixWithTrack:(AVMutableCompositionTrack *)
track {

    // To be implemented

    return nil;
}

@end

```

- 1.** You create a new instance of `AVMutableComposition` and begin adding `AVCompositionTrack` objects to it as you did in the previous chapter using the `addCompositionTrackOfType:withMediaItems:` method. This is the same method that you wrote in the previous chapter, but with a small modification to return a reference to the `AVMutableComposition` it creates.
- 2.** You build an instance of `AVAudioMix` calling the private `buildAudioMixWithTrack:` method that you'll be implementing shortly. This method is passed a reference to the track to which the volume adjustments should be applied. Finally, you create and return a new instance `THAudioMixComposition`, passing it the composition and audio mix.

Before we get into the implementation of the `buildAudioMixWithTrack:` method, I want to briefly discuss how the application represents the volume automation that is built in the user interface. The audio items contained in the `THTimeline` object are of type `THAudioItem`. This object provides a wrapper over the underlying `AVAsset` instance and additionally provides a property called `volumeAutomation`, which is an array of `THVolumeAutomation` instances. `THVolumeAutomation` is used to store the volume automation

data that is dynamically created in the user interface. It provides all the relevant data needed to create the volume parameters to be set on an instance of `AVMutableAudioMixInputParameters`. Let's look at the implementation of the `buildAudioMixWithTrack`: so you can see how this object is used (see [Listing 10.5](#)).

Listing 10.5 Building the Audio Mix

[Click here to view code image](#)

```
- (AVAudioMix *)buildAudioMixWithTrack:(AVCompositionTrack *)track
{
    THAudioItem *item = [self.timeline.musicItems
firstObject]; // 1
    if (item) {
        AVMutableAudioMix *audioMix = [AVMutableAudioMix
audioMix];
        // 2

        AVMutableAudioMixInputParameters *parameters =
            [AVMutableAudioMixInputParameters
                audioMixInputParametersWithTrack:track];

        for (THVolumeAutomation *automation in
item.volumeAutomation) { // 3
            [parameters
setVolumeRampFromStartVolume:automation.startVolume
                                toEndVolume:automation.endVolume
                                timeRange:automation.timeRange];
        }

        audioMix.inputParameters =
@parameters; // 4
        return audioMix;
    }
    return nil;
}
```

1. You begin by getting the `THAudioItem` instance for the music track from the timeline. The application allows only a single music track to be added, so you'll ask for the `firstObject` from the array of `musicItems`.
2. You create a new instance of `AVMutableAudioMix` that will be used to store the input parameters. You also create a new instance of `AVMutableAudioMixInputParameters`, associating it with the

`AVCompositionTrack` passed to this method.

3. You iterate over the objects in the audio item's `volumeAutomation` array, and for each instance, you define a volume ramp on the `parameters` object passing the `startVolume`, `endVolume`, and `timeRange`.
4. You wrap the `parameters` object in an `NSArray` and set it as the audio mix's `inputParameters` and return the audio mix instance.

Run the application. You can manually create a custom composition or simply tap the Settings menu and select Load Default Composition. A new Audio section has been added to the Settings menu containing two switches to toggle the audio processing behavior (see [Figure 10.4](#)).

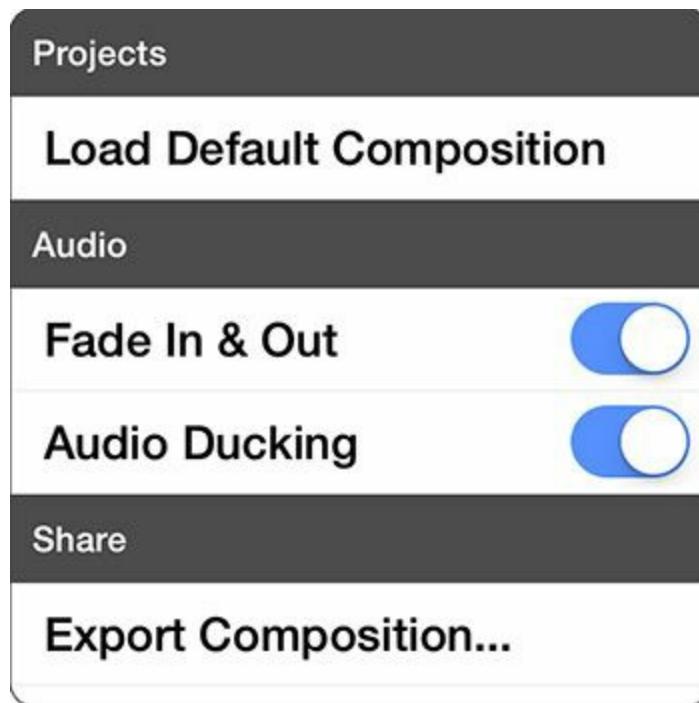


Figure 10.4 Settings menu—audio controls

Begin by enabling the Fade In & Out toggle switch and play the composition. You can see a volume curve has been drawn, providing a visual representation of what you should hear. The music now gently fades in and out and provides a much better experience for the listener. Now enable the Audio Ducking switch. When you do, you'll see the volume automation updated, providing a visual depiction of the ducking behavior. Play the composition again and you can hear that the voiceover comes though loud and clear. The volume automation data is dynamically created, so you can

slide the voiceover around and see and hear your changes.

Summary

In this chapter you learned how to use `AVAudioMix` and `AVAudioMixInputParameters` to provide greater control over your composition's audio tracks. The changes made to the 15 Seconds app were fairly small and easy to implement, but they provide a significant enhancement to the app. Audio mixing is an essential feature in any audio or video editing application, but you'll find that almost all uses of `AVComposition` can benefit from using `AVAudioMix` to polish your audio output.

Challenge

Experiment with customizing the volume automation the application produces. You'll find a method in the `THTimelineViewController` called `buildVolumeFadesForMusicItem`: that builds the volume automation data based on the changes made in the user interface. Change the duration of the audio fades, play with different volume levels for the audio ducking range, throw an arbitrary volume boost, or cut into the mix. It's your app, so experiment and have fun!

11. Building Video Transitions

The 15 Seconds app provides an easy way to quickly perform cuts-only video editing. As the video timeline progresses, each scene immediately transitions, or *cuts*, to the next with no transition effect between the two. This type of transition is seen most frequently, but it's not the only type you'll encounter. Often when there is a significant change in the timeline, such as a change in plot or location, some type of animated transition occurs between scenes. Video transitions such as fades, dissolves, and wipes are used for a number of creative effects and can help support a video production's story or message. In this chapter we explore AV Foundation's support for building video transitions and see how these features can help you add polish to your video editing applications.

Overview

An important feature of most video editing applications is the capability of creating animated video transitions. AV Foundation provides robust support in this regard, but it is arguably the most challenging part of the media editing APIs to learn. It's one of the least documented parts of the framework, involves the use of some slightly confusing API, and can be rather difficult to debug. This chapter provides a step-by-step guide to give you a solid understanding of the topic and help you avoid the most common pitfalls. Let's begin the process by looking at the classes used to build video transitions (see [Figure 11.1](#)).

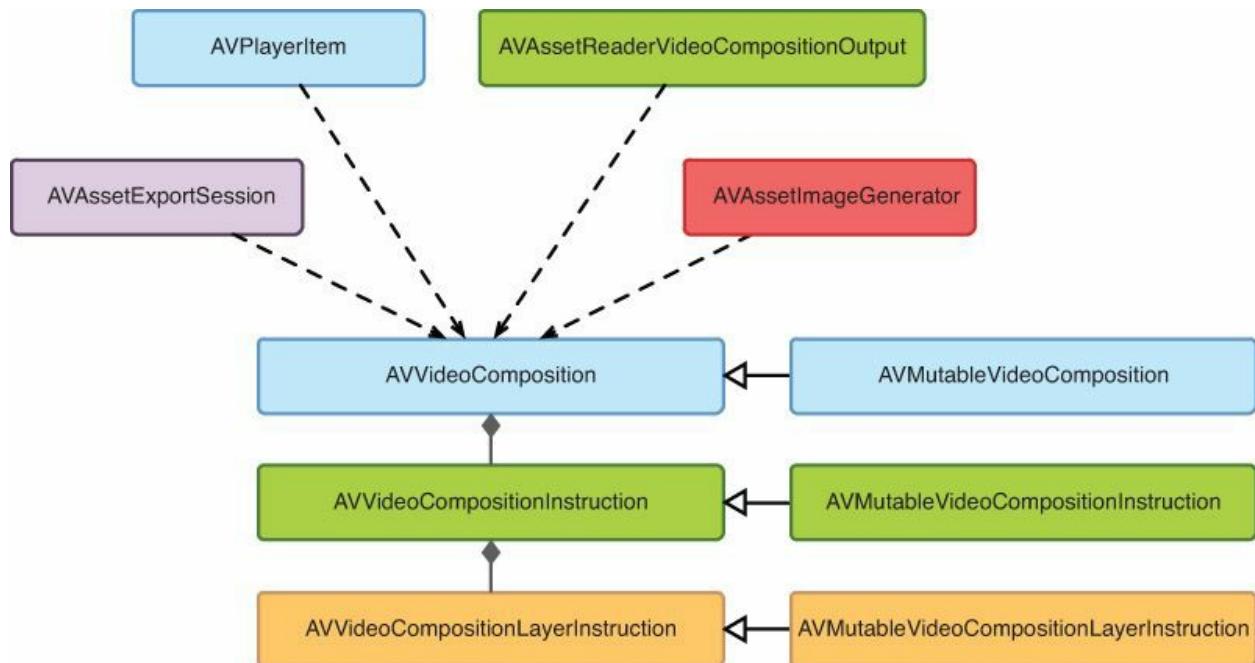


Figure 11.1 Video composition classes

AVVideoComposition

The central class in the framework's video transition APIs is `AVVideoComposition`. This class provides an aggregate description of the way two or more video tracks are composited together. It's composed of a collection of time ranges and instructions describing the compositing behavior that should occur at any point in time within the composition. In addition to the instructions it contains describing the compositing of its input video layers, it also provides properties to configure the video composition's rendering size, scale, and frame duration. The video composition's configuration determines the presentation of your `AVComposition` when processed by client objects such as `AVPlayer` or `AVAssetImageGenerator`.

Note

An important point to note is that despite its name, an `AVVideoComposition` is not a subclass of, or even directly related to, an `AVComposition`. It's used solely to control the video compositing behavior of an asset's video tracks when played, exported, or processed.

AVVideoCompositionInstruction

An AVVideoComposition is composed of a collection of instructions provided in the form of an object called

AVVideoCompositionInstruction. One of the key pieces of data this object provides is a time range within a composition's timeline over which some form of compositing should occur. The exact nature of the compositing to be performed is defined by its collection of layerInstructions.

AVVideoCompositionLayerInstruction

An AVVideoCompositionLayerInstruction is used to define the opacity, transform, and cropping effects applied to a given video track. It provides methods to modify these values at either a specific point in time or gradually over a time range. Applying ramps to these values over a time range enables you to create animated transition effects such dissolves and fades.

Like all AV Foundation media editing classes, the video composition APIs come in both immutable and mutable forms. The immutable superclass forms are suitable for use by their client objects, such as AVAssetExportSession, but when you are building your own video compositions you'll be working with their mutable subclasses.

Note

An AVVideoComposition, just like an AVAudioMix, is not directly associated with an AVComposition. Instead, these objects are associated with clients like AVPlayerItem and are used when the composition is played or otherwise processed. At first glance this may seem odd, but there are benefits to this design. Not strongly coupling an AVComposition to its output behavior provides you greater flexibility to determine how those behaviors should be applied when it is played, exported, or processed.

Conceptual Steps

In the course of developing the 15 Seconds app, you've been working with a single video track with a series of video clips arranged sequentially along its

timeline, as shown in [Figure 11.2](#).



Figure 11.2 Current video track layout

This track arrangement is precisely what is needed when building a cuts-only edit. However, it's insufficient when you need to perform animated transitions between the individual video segments. An effective way of learning to build transitions is to decompose the process into a series of steps. These are really *conceptual* steps that aid in understanding the process, but after you are comfortable with the concepts you will often combine two or more into a single unit of work.

1 Stagger the Video Layout

To enable transitions between clips, you first need to begin by staggering your video clip layout across two tracks. You've already seen how you can have multiple audio tracks in a composition, and the same is true for video tracks. In most cases two video tracks will suffice, but additional tracks can be added to meet the needs of your particular use case. Keep in mind that adding too many concurrent tracks can have an adverse impact on performance. You'll begin by creating a composition containing two video tracks, as depicted in [Figure 11.3](#).

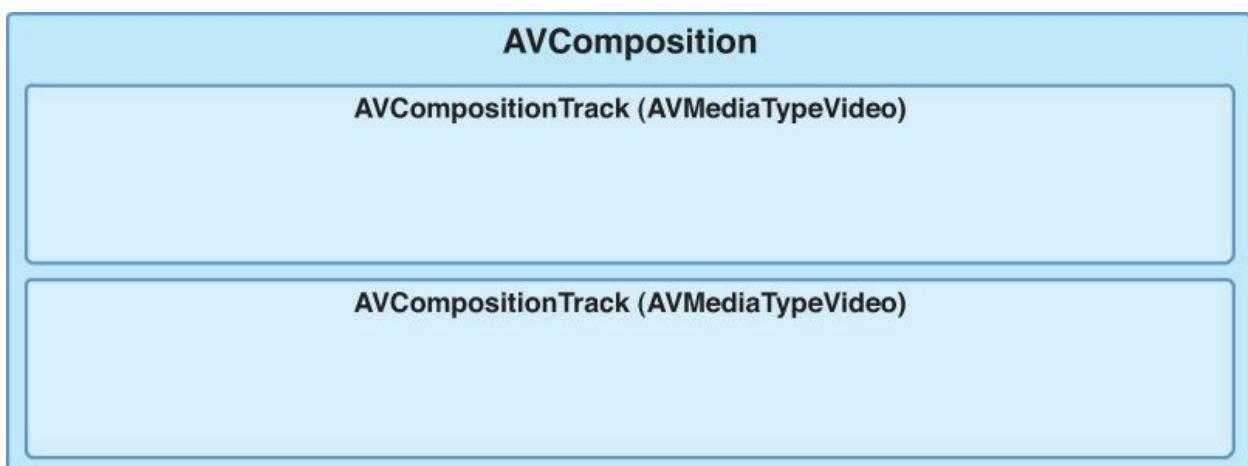


Figure 11.3 AVComposition with two video tracks

The code required to build the composition shown in the preceding figure would be written as follows.

[Click here to view code image](#)

```
AVMutableComposition *composition = [AVMutableComposition  
composition];  
  
AVMutableCompositionTrack *trackA =  
[composition addMutableTrackWithMediaType:AVMediaTypeVideo  
preferredTrackID:kCMPersistentTrackID];  
  
AVMutableCompositionTrack *trackB =  
[composition addMutableTrackWithMediaType:AVMediaTypeVideo  
preferredTrackID:kCMPersistentTrackID];  
  
NSArray *videoTracks = @[trackA, trackB];
```

The example begins by creating a new mutable composition and adding two composition tracks of type `AVMediaTypeVideo` to it. It adds the composition tracks to an `NSArray` for later use.

With the needed track arrangement in place, you then stagger the layout of the videos across both tracks, as shown in [Figure 11.4](#).

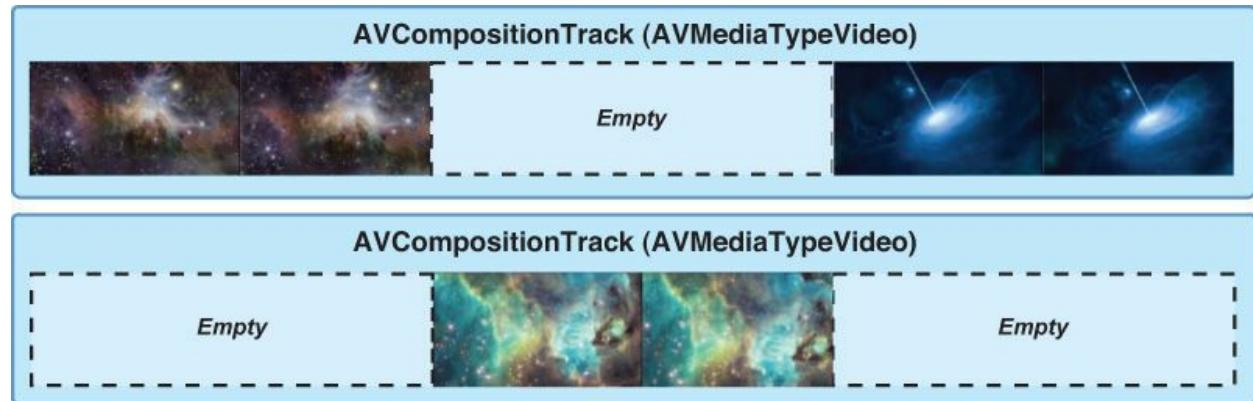


Figure 11.4 Building a staggered layout

[Figure 11.4](#) shows an arrangement of three videos, but you would follow this same A-B pattern for all video clips contained within your composition. When your video segments are arranged this way, any resulting gaps before or after a video segment will be filled with an *empty segment*. These are normal instances of `AVCompositionTrackSegment` just like the videos, but they contain no media. Following is an example of this in code.

[Click here to view code image](#)

```
NSArray *videoAssets = nil; // array of loaded AVAsset instances;
```

```

CMTIME cursorTime = kCMTimeZero;

for (NSUInteger i = 0; i < videoAssets.count; i++) {

    NSUInteger trackIndex = i % 2;

    AVMutableCompositionTrack *currentTrack =
videoTracks[trackIndex];

    AVAsset *asset = videoAssets[i];
    AVAssetTrack *assetTrack =
        [[asset tracksWithMediaType:AVMediaTypeVideo]
firstObject];

    CMTIMERange timeRange = CMTIMERangeMake(kCMTimeZero,
asset.duration);

    [currentTrack insertTimeRange:timeRange
                           ofTrack:assetTrack
                           atTime:cursorTime error:nil];

    cursorTime = CMTimeAdd(cursorTime, timeRange.duration);
}

```

The example iterates over the collection of video assets to be added to the composition. It begins each iteration by calculating the target track index using `i % 2`, resulting in the needed A-B pattern shown earlier. It extracts the video track from the current asset and inserts it into the `currentTrack`. Finally, it updates the `cursorTime` to update the insertion point for the next iteration of the loop.

The track and clip layout looks good, but one problem remains to be resolved. Can you spot it? Even though we've staggered the clips in an alternating fashion across two tracks, each segment still starts at the end of the previous segment. This means we don't have any regions defined where both clips will be visible and can be composited together. Let's resolve this in the next step.

2 Define Overlapping Regions

To perform a video transition between segments, you need to make sure the segments overlap by the desired transition duration. This can be easily resolved by making a small adjustment to the way the `cursorTime` was calculated in the previous example.

[Click here to view code image](#)

```
NSArray *videoAssets = nil;// array of loaded AVAsset instances;

CMTime cursorTime = kCMTimeZero;
CMTime transitionDuration = CMTimeMake(2, 1);

for (NSUInteger i = 0; i < videoAssets.count; i++) {

    NSUInteger trackIndex = i % 2;

    AVMutableCompositionTrack *currentTrack =
videoTracks[trackIndex];

    AVAsset *asset = videoAssets[i];
    AVAssetTrack *assetTrack =
        [[asset tracksWithMediaType:AVMediaTypeVideo]
firstObject];

    CMTimeRange timeRange = CMTimeRangeMake(kCMTimeZero,
asset.duration);

    [currentTrack insertTimeRange:timeRange
                           ofTrack:assetTrack
                           atTime:cursorTime error:nil];

    cursorTime = CMTimeAdd(cursorTime, timeRange.duration);
    cursorTime = CMTimeSubtract(cursorTime,
transitionDuration);
}
```

The example defines a CMTime value for the desired transition duration, which in this case is 2 seconds. At the end of the loop it subtracts the transitionDuration from the cursorTime so the next insertion point will be offset appropriately to account for the transition duration. [Figure 11.5](#) shows the result of this change.

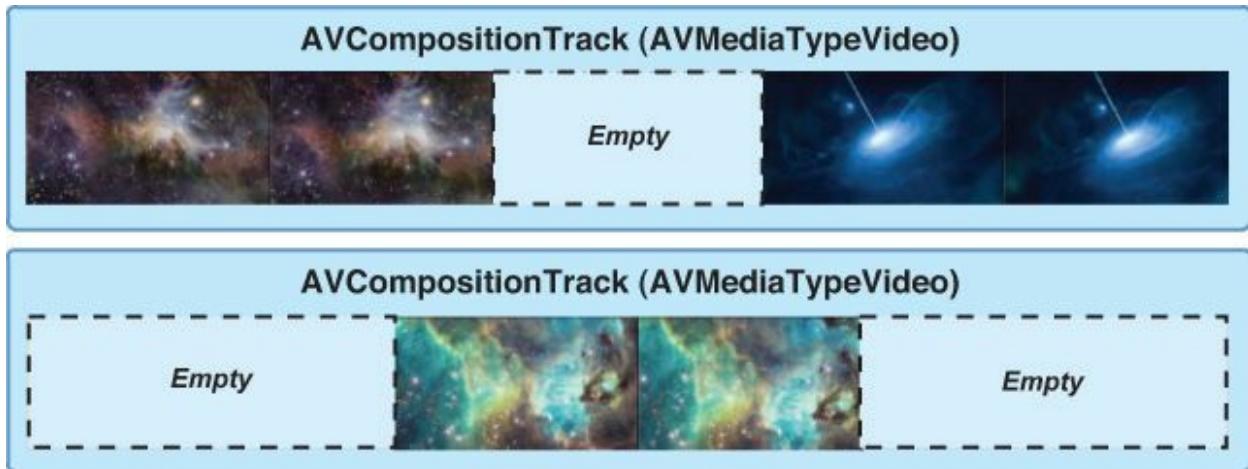


Figure 11.5 Resulting overlapping regions

The composition is playable at this point; however, the resulting output will not be as desired. Video tracks have an inherent z-indexing behavior where the first track sits in front of the second and the second in front of the third, and so on. If you played the composition as it currently stands, you'd see the first clip, followed by nothing for the second, and finally you'd see the third clip as it's contained within the front-most track. Before you can see the content of the second track, you need to define the time ranges and instructions to explain to the compositor how the tracks should be composited. Let's move on to the next step and begin that process.

Note

Keep in mind that by creating these overlapping regions, you have shortened the video timeline by $(videoCount - 1) * transitionDuration$. If you have additional tracks in your composition, you may need to shorten their duration to fit the new video timeline.

3 Calculate Pass-Through and Transition Time Ranges

An AVVideoComposition is composed of a collection of AVVideoCompositionInstruction objects. The most important piece of data this instruction contains is a time range, which defines a time range over which some form of compositing should occur. Before you can begin building instances of AVVideoCompositionInstruction, you first need to compute a series of time ranges for the composition.

You'll need to calculate two types of time ranges. The first is commonly

referred to as a *pass-through* time range. Over this time range you want the full series of frames of one track to pass through without any blending with the other. The other type of time range is a *transition* time range. This defines a time range in your composition where the video segments overlap and marks a region in the timeline where a transition effect will be applied. [Figure 11.6](#) shows the pass-through and transition time ranges in our example composition.

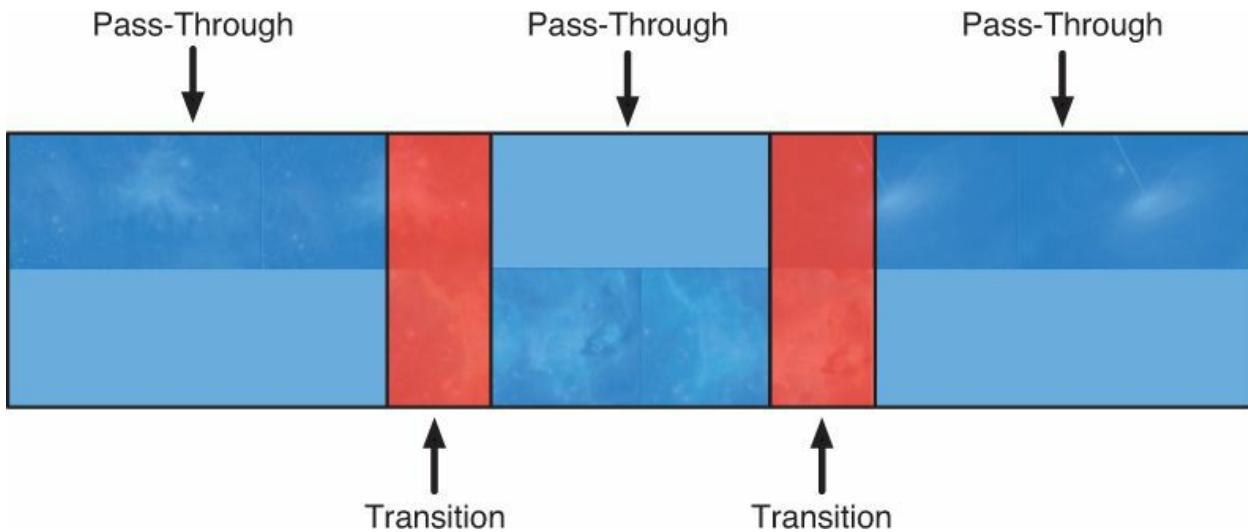


Figure 11.6 Pass-through and transition time ranges

You could calculate these time ranges in a number of ways, but the following code example provides a good general-purpose way of accomplishing this. This process is often performed as part of the track and segment building process, but is being done as a separate step in this example for simplicity's sake.

[Click here to view code image](#)

```

NSMutableArray *videoAssets = // loaded video assets

CMTime cursorTime = kCMTimeZero;

// 2 second transition duration
CMTime transDuration = CMTimeMake(2, 1);

NSMutableArray *passThroughTimeRanges = [NSMutableArray array];
NSMutableArray *transitionTimeRanges = [NSMutableArray array];

NSUInteger videoCount = [videoAssets count];

for (NSUInteger i = 0; i < videoCount; i++) {

```

```

AVAsset *asset = videoAssets[i];

CMTIMERange timeRange = CMTIMERangeMake(cursorTime,
asset.duration);

if (i > 0) {
    timeRange.start = CMTimeAdd(timeRange.start,
transDuration);
    timeRange.duration = CMTimeSubtract(timeRange.duration,
transDuration);
}

if (i + 1 < videoCount) {
    timeRange.duration = CMTimeSubtract(timeRange.duration,
transDuration);
}

[passThroughTimeRanges addObject:[NSValue
valueWithCMTimeRange:timeRange]];

cursorTime = CMTimeAdd(cursorTime, asset.duration);
cursorTime = CMTimeSubtract(cursorTime, transDuration);

if (i + 1 < videoCount) {
    timeRange = CMTIMERangeMake(cursorTime, transDuration);
    NSValue *timeRangeValue = [NSValue
valueWithCMTimeRange:timeRange];
    [transitionTimeRanges addObject:timeRangeValue];
}

}

```

The example iterates through the collection of the composition's videos. For each video it builds an initial time range, and then depending on its ordinal position, makes some modifications to the time range's start and/or duration times. After the `cursorTime` value has been calculated, it builds the related transition time range based on the `cursorTime` and `transDuration`.

Note

`CMTIME` and `CMTIMERange` are structs, so they can't be directly added to an `NSArray` or an `NSDictionary`. Instead, you'll wrap them in instances of `NSValue` using the category methods defined by `AVTime.h`.

Tip

The effects of the preceding example can be difficult to describe in text. I have found drawing boxes on a sheet of paper as you step through the code can help you to better visualize the process.

Building the required pass-through and transition time ranges is arguably the most important step in this process. On its face, it doesn't seem like a particularly hard problem to solve; however, it's surprisingly easy to get wrong. There are two key points to keep in mind when performing this step:

- The time ranges you calculate must not have any gaps or overlaps. They must be laid out sequentially with each new time range beginning at the end of the previous one.
- Your calculations must take the total *composition duration* into consideration. If you have additional tracks in your composition, you'll need to either fit them to the video timeline or extend your final time range to account for their duration.

If you fail to observe either of those two points, your composition will still play, but your video content will not be rendered, leaving you staring at a black screen. This can be frustrating to debug, but Apple has released a utility class that can help you better diagnose these problems. On the Apple Developer Center you'll find a project called **AVCompositionDebugViewer** (available in both Mac and iOS versions) that will help you visualize your composition. [Figure 11.7](#) provides a screenshot of this utility in action.

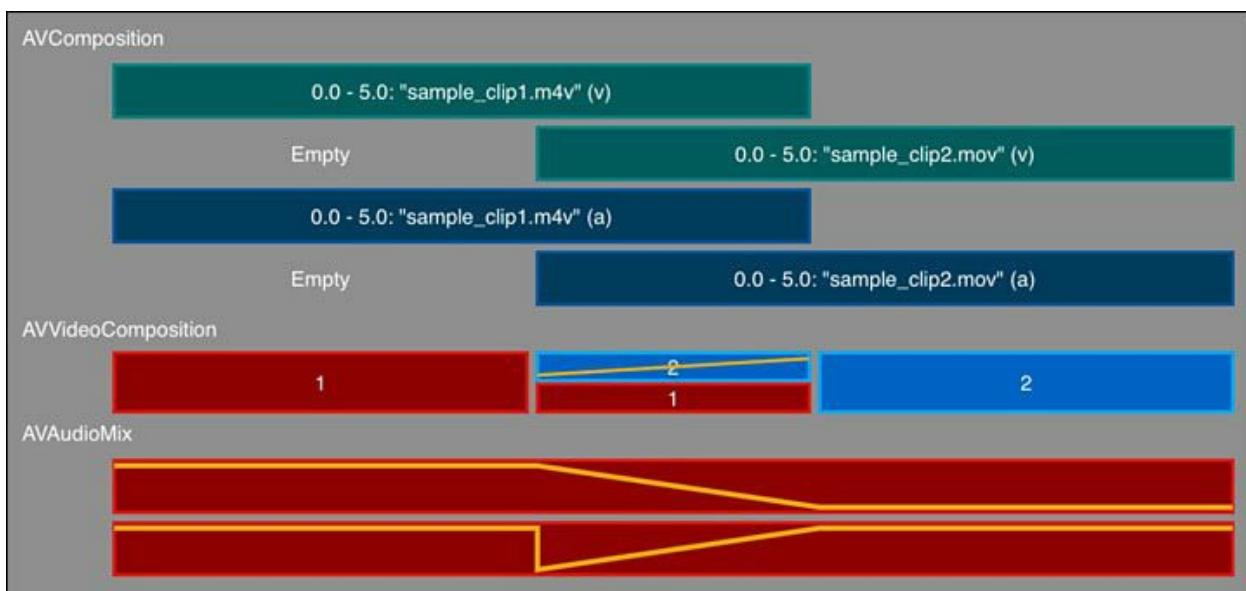


Figure 11.7 AVCompositionDebugViewer

This class helps you visualize your composition and its associated `AVVideoComposition` and `AVAudioMix` objects, making it easier to identify problems.

Now that we've built the required pass-through and transition time ranges, it's time to move on to the next step and build the compositing instructions.

4 Build Composition and Layer Instructions

The next step is to build the instances of `AVVideoCompositionInstruction` and `AVVideoCompositionLayerInstruction` that provide the instructions to be performed by the video compositor. The following code example is fairly involved, so it will be followed by an annotated breakdown.

[Click here to view code image](#)

```
NSMutableArray *compositionInstructions = [NSMutableArray array];

// Look up all of the video tracks in the composition
NSArray *tracks = [composition
tracksWithMediaType:AVMediaTypeVideo];

for (NSUInteger i = 0; i < passThroughTimeRanges.count; i++)
{
    // 1

    // Calculate the trackIndex to operate upon: 0, 1, 0, 1, etc.
    NSUInteger trackIndex = i % 2;

    AVMutableCompositionTrack *currentTrack = tracks[trackIndex];

    AVMutableVideoCompositionInstruction *instruction
    =           // 2
        [AVMutableVideoCompositionInstruction
videoCompositionInstruction];

    instruction.timeRange =  [passThroughTimeRanges[i]
CMTIMEVALUE];

    AVMutableVideoCompositionLayerInstruction *layerInstruction
    =           // 3
        [AVMutableVideoCompositionLayerInstruction
videoCompositionLayerInstructionWithAssetTrack:current];

    instruction.layerInstructions = @[layerInstruction];
```

```

[compositionInstructions addObject:instruction];

if (i < transitionTimeRanges.count) {

    AVCompositionTrack *foregroundTrack =
tracks[trackIndex]; // 4
    AVCompositionTrack *backgroundTrack = tracks[1 -
trackIndex];

    AVMutableVideoCompositionInstruction *instruction
= // 5
        [AVMutableVideoCompositionInstruction
videoCompositionInstruction];

    CMTIMERange timeRange = [transitionTimeRanges[i]
CMTIMERangeValue];
    instruction.timeRange = timeRange;

    AVMutableVideoCompositionLayerInstruction
*fromLayerInstruction = // 6
        [AVMutableVideoCompositionLayerInstruction
videoCompositionLayerInstructionWithAssetTrack:fore-
groundTrack];

    AVMutableVideoCompositionLayerInstruction
*toLayerInstruction =
        [AVMutableVideoCompositionLayerInstruction
videoCompositionLayerInstructionWithAssetTrack:bac-
kgroundTrack];

    instruction.layerInstructions =
@[fromLayerInstruction, // 7
    toLayerInstruction];

    [compositionInstructions addObject:instruction];
}

}

```

- 1.** The example begins by iterating over all the pass-through time ranges calculated earlier. The loop toggles back and forth between the two video tracks to build the required instructions for each.
- 2.** It creates a new instance of `AVMutableVideoCompositionInstruction` and sets the current pass-through `CMTIMERange` as its `timeRange` property.
- 3.** The example then builds an `AVMutableVideoCompositionLayerInstruction` for the active composition track, adds it to an array, and then sets it as the

composition instruction's `layerInstructions` property. The pass-through regions of the composition require only a single layer instruction associated with the track whose video frames should be presented.

4. To build the transition time range instructions, it gets a reference to the foreground track (the one that is being transitioned from) and the background track (the one that is being transitioned to). Performing the track lookup in the manner shown ensures the track references are always in the correct order.
5. It builds another instance of `AVMutableVideoCompositionInstruction`, setting the current transition time range as its `timeRange` property.
6. It creates an instance of `AVMutableVideoCompositionLayerInstruction` for each track. On these layer instructions you define the transitioning effects describing how you transition from one to the other. No transition effects are being applied in this example. Instead, we'll defer that discussion until we get into this chapter's sample application.
7. It adds both of the layer instructions to an `NSArray` and sets them as the current composition instruction's `layerInstructions` property. The ordering of the elements in this array is important because it defines the z-axis ordering of the video layers in the composited output.

All the required composition and layer instructions have been successfully created, and it's time to move on to the final step, where we'll build and configure the `AVVideoComposition`.

5 Build and Configure the `AVVideoComposition`

The hard work has been completed and all that remains is to build and configure an instance of `AVVideoComposition` as shown in the following example:

[Click here to view code image](#)

```
AVMutableVideoComposition *videoComposition =
    [AVMutableVideoComposition videoComposition];
videoComposition.instructions = compositionInstructions;
videoComposition.renderSize = CGSizeMake(1280.0f, 720.0f);
```

```
videoComposition.frameDuration = CMTimeMake(1, 30);  
videoComposition.renderScale = 1.0f;
```

The example builds an instance of `AVMutableVideoComposition` and sets its four key properties:

- The `instructions` property is set to the composition instructions that were created in step 4. These instructions describe to the compositor the time range and nature of the compositing to be performed.
- The `renderSize` property is a `CGSize` value defining the size at which this composition should be rendered. The value provided should correspond to the natural size of the videos contained within the composition, such as 1280 x 720 for 720p video or 1920 x 1080 for 1080p content.
- The `frameDuration` is used to set the effective *frame rate*. Recall that AV Foundation rarely deals in frame rates, but instead uses frame durations. The frame duration is the reciprocal of the frame rate, so to set a frame rate of 30FPS, you would define a frame duration of 1/30th of a second.
- The `renderScale` defines the scaling applied to the video composition. In most cases this will be set to a value of 1.0.

If this is the first time you have seen how to build video transitions in AV Foundation, it's a relatively safe assumption that you've sought out some aspirin or a stiff drink by now. I feel your pain. Learning to build video transitions is certainly the most complex portion of the framework's video editing capabilities, but I do have a bit of good news to tell you, so read on.

Building Video Compositions the Auto-Magic Way

In many cases there is an easier way of building an `AVVideoComposition` than what was described in the previous section. `AVVideoComposition` provides a convenience initializer called `videoCompositionWithPropertiesOfAsset:` that takes your `AVComposition` as the asset argument and builds a base `AVVideoComposition` for you. This method creates an `AVVideoComposition` with the following configuration:

- The `instructions` property contains a complete set of composition

and layer instructions based on the composition's video tracks and the spatial layout of the segments contained in each.

- The `renderSize` property is set to the `AVComposition` object's `naturalSize`, or if not set, a size large enough to fit the greatest video dimensions contained within the composition's video tracks.
- The `frameDuration` is set to a value accommodating the greatest `nominalFrameRate` found in the composition's video tracks. If the `nominalFrameRate` value for all tracks is 0, the `frameDuration` will be set to a default value 1/30th of a second (30FPS).
- The `renderScale` is always set to 1 . 0.

You may be wondering why I didn't lead with this approach and skip all the complicated stuff. Well, there are actually two important reasons:

- Unless you understand the steps discussed earlier, you'll find yourself struggling to understand how to make use of this object, because you will still need to configure the instructions it provides to define your transition effects. Failing to have a clear understanding of how and why the composition is constructed the way it is will make it difficult for you to get the results you need and will make it nearly impossible to debug when things go wrong.
- What it builds may not be what you need. Building an `AVVideoComposition` this way can be useful in many cases, but it's not a one-size-fits-all solution. There may be times when you need more control over the composition's time ranges and instructions, and in those cases it's still necessary to manually build the composition from scratch.

Ultimately, the more you understand about this process, the more quickly you can become proficient using AV Foundation's transition capabilities.

15 Seconds: Adding Video Transitions

Now that you are familiar with the concepts used when building video transitions, let's discuss how to enable this functionality in the 15 Seconds app. You'll find a starter version of the 15 Seconds project in the [Chapter 10](#) directory that picks up where you left off in the previous chapter.

You begin by creating a new class adopting the `THComposition` protocol

called `THTransitionComposition` (see [Listing 11.1](#)).

Listing 11.1 **THTransitionComposition** Interface

[Click here to view code image](#)

```
#import "THComposition.h"

@interface THTransitionComposition : NSObject <THComposition>

@property (strong, nonatomic, readonly) AVComposition
*composition;
@property (strong, nonatomic, readonly) AVVideoComposition
*videoComposition;
@property (strong, nonatomic, readonly) AVAudioMix *audioMix;

- (id)initWithComposition:(AVComposition *)composition
    videoComposition:(AVVideoComposition *)videoComposition
    audioMix:(AVAudioMix *)audioMix;
@end
```

This class looks similar to the ones you've built over the past couple chapters; the only change is the addition of a new `AVVideoComposition` property and a `videoComposition` argument added to the initializer. Let's look at the implementation of this class in [Listing 11.2](#).

Listing 11.2 **THTransitionComposition** Implementation

[Click here to view code image](#)

```
#import "THTransitionComposition.h"

@implementation THTransitionComposition

- (id)initWithComposition:(AVComposition *)composition
    videoComposition:(AVVideoComposition *)videoComposition
    audioMix:(AVAudioMix *)audioMix {
    self = [super init];
    if (self) {
        _composition = composition;
        _videoComposition = videoComposition;
        _audioMix = audioMix;
    }
    return self;
}

- (AVPlayerItem *)makePlayable
```

```

{
    // 1
    AVPlayerItem *playerItem =
        [AVPlayerItem playerItemWithAsset:[self.composition
copy]];
}

playerItem.audioMix = self.audioMix;
playerItem.videoComposition = self.videoComposition;

return playerItem;
}

- (AVAssetExportSession *)makeExportable
{
    // 2
    NSString *preset = AVAssetExportPresetHighestQuality;
    AVAssetExportSession *session =
        [AVAssetExportSession exportSessionWithAsset:
    [self.composition copy]
                                presetName:preset];
    session.audioMix = self.audioMix;
    session.videoComposition = self.videoComposition;

    return session;
}

@end

```

- 1.** You begin the `makePlayable` method by creating a new `AVPlayerItem` with the `AVComposition` instance. You set the player item's `audioMix` as you did in the previous chapter, and you also set its `videoComposition` property. This enables both the custom audio processing and video compositing behavior to be applied to this player item when played.
- 2.** You begin the `makeExportable` method by creating a new `AVAssetExportSession`, passing it a copy of the `composition` and a preset name. Similar to the `makePlayable` method, you'll set the `audioMix` and `videoComposition` on the export session's respective properties so the audio and video processing will be applied on export.

The next step is to create the associated `THCompositionBuilder` for this object. You'll find a stubbed version of a class called `THTransitionCompositionBuilder` in the sample project under the `Models/Builder/Private` group. [Listing 11.3](#) provides the interface

for this class.

Listing 11.3 THTransitionCompositionBuilder Interface

[Click here to view code image](#)

```
#import "THCompositionBuilder.h"
#import "THTimeline.h"

@interface THTransitionCompositionBuilder : NSObject
<THCompositionBuilder>

- (id)initWithTimeline:(THTimeline *)timeline;

@end
```

With the exception of the class name, this interface looks like all the others you've built over the past couple of chapters, so let's move on and begin implementing its behavior (see [Listing 11.4](#)).

Listing 11.4 THTransitionCompositionBuilder Implementation

[Click here to view code image](#)

```
#import "THTransitionCompositionBuilder.h"
#import "THVideoItem.h"
#import "THAudioItem.h"
#import "THVolumeAutomation.h"
#import "THTransitionComposition.h"
#import "THTransitionInstructions.h"
#import "THFunctions.h"

@interface THTransitionCompositionBuilder ()
@property (strong, nonatomic) THTimeline *timeline;
@property (strong, nonatomic) AVMutableComposition *composition;
@property (weak, nonatomic) AVMutableCompositionTrack *musicTrack;
@end

@implementation THTransitionCompositionBuilder

- (id)initWithTimeline:(THTimeline *)timeline {
    self = [super init];
    if (self) {
        _timeline = timeline;
    }
    return self;
}
```

```

- (id <THComposition>)buildComposition {
    self.composition = [AVMutableComposition composition];
    [self buildCompositionTracks];
    AVVideoComposition *videoComposition = [self
buildVideoComposition];
    AVAudioMix *audioMix = [self buildAudioMix];
    return [[THTransitionComposition alloc]
initWithComposition:self.composition
                           videoComposition:videoComposition
                           audioMix:audioMix];
}

- (void)buildCompositionTracks {
    // To be implemented
}

- (AVVideoComposition *)buildVideoComposition {
    // To be implemented
    return nil;
}

@end

```

The implementation of the `buildComposition` methods looks similar to what you built in the previous chapter, but the track building code has been factored out into its own method, and a new method called `buildVideoComposition` has been added—both of which you’ll be implementing shortly. The implementation of the `buildAudioMix` method is omitted because it remains unchanged from the previous chapter.

Let’s implement the `buildCompositionTracks` method to provide the track and clip layout required to enable video transitions (see [Listing 11.5](#)).

Listing 11.5 Implementing the `buildCompositionTracks` Method

[Click here to view code image](#)

```
- (void)buildCompositionTracks {
```

```

    CMPersistentTrackID trackID = kCMPersistentTrackID_Invalid;

    AVMutableCompositionTrack *compositionTrackA
    =
        [self.composition
    addMutableTrackWithMediaType:AVMediaTypeVideo
                                preferredTrackID:trackID];

    AVMutableCompositionTrack *compositionTrackB =
        [self.composition
    addMutableTrackWithMediaType:AVMediaTypeVideo
                                preferredTrackID:trackID];

    NSArray *videoTracks = @ [compositionTrackA,
compositionTrackB];

    CMTIME cursorTime = kCMTIMEZero;
    CMTIME transitionDuration = kCMTIMEZero;

    if (!THIsEmpty(self.timeline.transitions))
    {
        // 2
        // 1 second transition duration
        transitionDuration = THDefaultTransitionDuration;
    }

    NSArray *videos = self.timeline.videos;

    for (NSUInteger i = 0; i < videos.count; i++) {

        NSUInteger trackIndex = i %
2;                                         // 3

        THVideoItem *item = videos[i];
        AVMutableCompositionTrack *currentTrack =
videoTracks[trackIndex];

        AVAssetTrack *assetTrack =
            [[item.asset tracksWithMediaType:AVMediaTypeVideo]
firstObject];

        [currentTrack insertTimeRange:item.timeRange
                                ofTrack:assetTrack
                                atTime:cursorTime error:nil];

        // Overlap clips by transition
duration                                         // 4
        cursorTime = CMTIMEAdd(cursorTime,
item.timeRange.duration);
        cursorTime = CMTimeSubtract(cursorTime,

```

```

transitionDuration);
}

// Add voice
overs // 5
[self addCompositionTrackOfType:AVMediaTypeAudio
    withMediaItems:self.timeline.voiceOvers];

// Add music track
NSArray *musicItems = self.timeline.musicItems;
self.musicTrack = [self
addCompositionTrackOfType:AVMediaTypeAudio
    withMediaItems:musicItems];
}

```

- 1.** Begin by creating two new `AVMutableCompositionTrack` objects, both of type `AVMediaTypeVideo`, to provide the needed A-B track arrangement. You store them in a `videoTracks` array so they can be retrieved while iterating over the collection of videos to be inserted into the composition.
- 2.** Determine if the `THTimeline` object's `transitions` array is populated. If video transitions have been enabled in the user interface, this array will be populated with a collection of `THVideoTransition` objects describing the transition effect to be applied. If transitions are enabled, you'll set the `transitionDuration` to `THDefaultTransitionDuration`, which is a global constant defining a 1 second `CMTime` value.
- 3.** Loop through the collection of videos to be added to the composition by first determining the target track into which the clip should be inserted. Computing the `trackIndex` as `i % 2` results in a 0, 1, 0, 1 pattern ensuring the appropriate A-B arrangement of your video segments.
- 4.** Calculate the `cursorTime` as you have done previously, but redefine its value using the `CMTimeSubtract` function to back up the insertion point to take the `transitionDuration` into an account. This ensures your track layout provides the needed transition regions.
- 5.** Add the voiceover and music tracks by calling the `addCompositionTrackOfType:withMediaItems:` method you wrote in a previous chapter. Its implementation is omitted from this

chapter for brevity's sake.

I recommend running the application at this point so you can see the result of this new composition arrangement. The videos contained in `compositionTrackA` will be presented as they always have, but the video contained in `compositionTrackB` will not be visible. This is due to the z-indexing behavior mentioned earlier when multiple video tracks are used. To fix this behavior you'll need to build an `AVVideoComposition` so you can control the compositing used when the composition is presented or exported. Let's begin building the `buildVideoComposition` in [Listing 11.6](#).

Listing 11.6 Implementing the `buildVideoComposition` Method

[Click here to view code image](#)

```
- (AVVideoComposition *)buildVideoComposition {
    AVVideoComposition *videoComposition
    // 1
    = [AVMutableVideoComposition
        videoCompositionWithPropertiesOfAsset:self.composition];

    NSArray *transitionInstructions
    // 2
    = [self
        transitionInstructionsInVideoComposition:videoComposition];

    for (THTransitionInstructions *instructions in
        transitionInstructions) {

        CMTimeRange timeRange
        // 3
        = instructions.compositionInstruction.timeRange;

        AVMutableVideoCompositionLayerInstruction *fromLayer =
            instructions.fromLayerInstruction;

        AVMutableVideoCompositionLayerInstruction *toLayer =
            instructions.toLayerInstruction;

        THVideoTransitionType type = instructions.transition.type;

        // Apply Video Transition
        Effects // 4
        if (type == THVideoTransitionTypeDissolve) {
            // Listing 11.7
```

```

    }

    else if (type == THVideoTransitionTypePush) {
        // Listing 11.8
    }

    else (type == THVideoTransitionTypeWipe) {
        // Listing 11.9
    }

    instructions.compositionInstruction.layerInstructions =
@[fromLayer, // 5
    ...
}

return videoComposition;
}

- (NSArray *)transitionInstructionsInVideoComposition:
(AVVideoComposition *)vc {

    // To be implemented

    return nil;
}

```

1. Begin by building a new instance of `AVVideoComposition` using the `videoCompositionWithPropertiesOfAsset:` method. This method automatically builds the needed composition and layer instructions and sets the `renderSize`, `renderScale`, and `frameDuration` to their appropriate values.

2. Call the private `transitionInstructionsInVideoComposition:` method to extract the relevant layer instructions from the `AVVideoComposition` so you can apply the desired video transition effects to them. This method returns an array of `THTransitionInstructions` objects, which are basic data holder objects used to simplify working with the composition and layer instructions. You'll provide an implementation of this method and build these objects shortly.

3. Define local variables for the data pulled from the `THTransitionInstructions` instance. These variables are used when applying the transition effects.

4. Define a placeholder conditional statement determining the video transition to perform. The application’s user interface enables you to apply a Dissolve, Push, or Wipe transition between clips. We cover the details of how those effects are created in [Listings 11.7–11.9](#).
5. Configure the composition instruction’s `layerInstructions`, passing the instructions in the order shown. This ordering is necessary to ensure the video transitions are applied as desired.

The next step is to provide an implementation for the `transitionInstructionsInVideoComposition:` method (see [Listing 11.7](#)). This method is responsible for extracting the composition and layer instructions out of the prebuilt `AVVideoComposition` and associating them with the `THVideoTransition` the user configured in the timeline. The `THVideoTransition` indicates the type of transition to be applied.

Listing 11.7 Extracting the Transition Instructions

[Click here to view code image](#)

```
- (NSArray *)transitionInstructionsInVideoComposition:
(AVVideoComposition *)vc {
    NSMutableArray *transitionInstructions = [NSMutableArray
array];
    int layerInstructionIndex = 1;
    NSArray *compositionInstructions =
vc.instructions; // 1
    for (AVMutableVideoCompositionInstruction *vci in
compositionInstructions) {
        if (vci.layerInstructions.count == 2)
{ // 2
            THTransitionInstructions *instructions =
[[THTransitionInstructions alloc] init];
            instructions.compositionInstruction = vci;
            instructions.fromLayerInstruction
// 3
= vci.layerInstructions[1 - layerInstructionIndex];
        }
    }
}
```

```

instructions.toLayerInstruction =
    vci.layerInstructions[layerInstructionIndex];

[transitionInstructions addObject:instructions];

layerInstructionIndex = layerInstructionIndex == 1 ? 0
: 1;
}

NSArray *transitions = self.timeline.transitions;

// Transitions are disabled
if (THIsEmpty(transitions))
{
    return transitionInstructions; // 4
}

NSAssert(transitionInstructions.count == transitions.count,
         @"Instruction count and transition count do not
match.");

for (NSUInteger i = 0; i < transitionInstructions.count; i++)
{
    // 5
    THTransitionInstructions *tis = transitionInstructions[i];
    tis.transition = self.timeline.transitions[i];
}

return transitionInstructions;
}

```

- 1.** Begin by iterating over the `AVVideoCompositionInstruction` objects retrieved from the `AVVideoComposition`.
- 2.** You're interested only in the composition instructions containing two layer instructions, which indicates this instruction defines a transition region in the composition.
- 3.** The layer instructions that are automatically built are always stored in the `layerInstructions` array with the first track's layer instructions first, followed by the second track's layer instructions. This code performs some calculations to ensure that the layer you are transitioning *from* and the layer you are transitioning *to* are always consistent. This makes it easier to think about the effects to be applied to each layer when building your transitions.

4. If the transitions array is empty, you simply return the transitionInstructions, because this indicates transitions have been disabled in the user interface.
5. If the transitions are enabled, you iterate over the transitionInstructions and associate the user selected THVideoTransition object with it. The THVideoTransition object defines the type of transition to be applied, such as push, dissolve, or wipe.

Applying Transition Effects

The application supports three types of transitions: Dissolve (the default), Push, and Wipe. Let's look at how to implement each of these, starting with the dissolve transition.

Dissolve Transition

The first transition to implement is a *dissolve* transition. This transition involves modifying the opacity of the input layers to achieve a gradual blending of the two (see [Figure 11.8](#)).



Figure 11.8 Dissolve transition

Let's see how this is implemented in [Listing 11.8](#).

Listing 11.8 Applying a Dissolve Transition

[Click here to view code image](#)

```
if (type == THVideoTransitionTypeDissolve) {  
    [fromLayer setOpacityRampFromStartOpacity:1.0  
     toEndOpacity:0.0
```

```
    timeRange:timeRange];  
}
```

Implementing a simple dissolve transition is easy. You simply set an opacity ramp on the `fromLayer` to adjust its opacity from its default of 1.0 (fully opaque) to 0.0 (fully transparent) over the transition duration. This has the effect of dissolving the current video into the next. An alternate form of a dissolve transition is called a *cross dissolve*, in which you'll also set an opacity ramp on the `toLayer`, ramping its opacity from 0.0 to 1.0. A simple dissolve is more subtle and typically looks better, but I encourage you to try out both forms and see which one you like best.

Push Transition

The next transition to implement is a *push* transition (see [Figure 11.9](#)). This is a directional transition where the next video “pushes” the current video out of view.



Figure 11.9 Push transition

[Listing 11.9](#) shows how to perform a *push left* effect.

Listing 11.9 Applying a Push Transition

[Click here to view code image](#)

```
else if (type == THVideoTransitionTypePush) {  
  
    // Define starting and ending  
    transforms                      // 1  
    CGAffineTransform identityTransform =  
    CGAffineTransformIdentity;  
  
    CGFloat videoWidth = videoComposition.renderSize.width;
```

```

CGAffineTransform fromDestTransform =
    CGAffineTransformMakeTranslation(-videoWidth, 0.0);

CGAffineTransform toStartTransform =
    CGAffineTransformMakeTranslation(videoWidth, 0.0);

[fromLayer
setTransformRampFromStartTransform:identityTransform // 2
toEndTransform:fromDestTransform
timeRange:timeRange];

[toLayer
setTransformRampFromStartTransform:toStartTransform // 3
toEndTransform:identityTransform
timeRange:timeRange];
}

```

- 1.** Begin by defining the transforms to be applied to the input video layers. A `CGAffineTransform` enables you to modify a layer's translation, rotation, and scaling. Applying a transform ramp to a layer enables you to perform a number of interesting effects. For this transition you'll be applying a *translation* transform, which enables you to modify a layer's x, y coordinates. The `fromDestTransform` will move the `fromLayer` to the left, by the negative video width, which will move it completely out of view. You define a `toStartTransform` to move the starting location of the `toLayer` out of view to the right.
- 2.** Set a transform ramp on the `fromLayer` passing the `identityTransform` (its untransformed state) as the starting transform and the `fromDestTransform` as its ending transform. This results in the `fromLayer` pushing left in an animated manner over the `timeRange`.
- 3.** You likewise set a transform ramp on the `toLayer` passing the `toStartTransform` as the starting transform and the `identityTransform` as its ending transform. This will cause the `toLayer` to animate in from the right over the `timeRange` duration.

Wipe Transition

Finally, the last transition to implement is a *wipe* (see [Figure 11.10](#)). Wipe

transitions come in a variety of forms, but they typically move the current video out in an animated manner, revealing the next video.



Figure 11.10 Wipe Up transition

[Listing 11.10](#) shows how to perform a simple *wipe up* where the `fromLayer` will wipe upward, revealing the `toLayer`.

Listing 11.10 Applying a Wipe Transition

[Click here to view code image](#)

```
else (type == THVideoTransitionTypeWipe) {  
  
    CGFloat videoWidth = videoComposition.renderSize.width;  
    CGFloat videoHeight = videoComposition.renderSize.height;  
  
    CGRect startRect = CGRectMake(0.0f, 0.0f, videoWidth,  
        videoHeight);  
    CGRect endRect = CGRectMake(0.0f, videoHeight, videoWidth,  
        0.0f);  
  
    [fromLayer  
    setCropRectangleRampFromStartCropRectangle:startRect  
                                toEndCropRectangle:endRect  
                                timeRange:timeRange  
    ]};
```

To achieve this effect, you retrieve the width and height from the video composition's `renderSize`. These values are used to create starting and ending `CGRect` values that perform an animated cropping effect. The starting rectangle is the full width and height, and the ending rectangle collapses its height, which produces a wiping up effect on the `fromLayer`.

Run the application and tap the Settings menu. You'll see that a new Video

section has been added to the menu with a Video Transitions toggle switch. Enable this switch and you'll see the video track update to reveal transition buttons, as shown in [Figure 11.11](#).



Figure 11.11 User interface transition control

Tap these buttons and experiment with the available transition types to see your work in action.

Summary

In this chapter you learned how to build video transitions using the `AVVideoComposition`, `AVVideoCompositionInstruction`, and `AVVideoCompositionLayerInstruction` objects. You gained a detailed understanding of how to build a video composition and learned some tips along the way that will help you avoid some common pitfalls. Learning to build video transitions can be challenging, but you now have a solid understanding of the process used to build transitions that will help you leverage the framework's features to add polish to your video editing applications.

Challenge

You added three common transitions to the 15 Seconds app, but you don't have to stop there. Change the direction of the push or wipe transitions. Implement a cross-dissolve between videos. You can even apply multiple ramps at the same time. Experimenting with these features is the best way to learn how to use them.

12. Layering Animated Content

The 15 Seconds app looks great and is capable of producing very professional-looking video compositions, but there is one remaining topic to address before bringing the book to a close. The application's track view has contained video, voiceover, and music tracks, but in all the compositions you've created so far there has always been one mystery track that has remained empty. It's time to clear up the mystery and discuss layering animated content over your compositions. This provides you with the capability of adding overlay effects, such as watermarks, titles, lower thirds, or any other animated effects you may need.

AV Foundation provides very good support for incorporating animated overlays in your video content. However, the way this is accomplished can seem a little strange at first. In all the work you've done so far you've been working in a track-oriented model, so it would seem reasonable to expect to find a specialized track type for incorporating your overlay effects, but this isn't the case. Instead, AV Foundation relies on Core Animation to build these effects and provides some integration classes that help incorporate them into your video content. This introduces a different mental model than the one you've been using over the past few chapters, but we'll discuss how to bridge that gap to provide you with a normalized way of working with AV Foundation and Core Animation. Let's begin our discussion of this topic by starting with a brief overview of Core Animation.

Using Core Animation

Core Animation is the compositing and animation framework provided on OS X and iOS that enables the beautiful, fluid animations seen on Apple platforms. It offers a simple, declarative programming model that makes it easy to build high-performance, GPU-backed animations without needing to turn to OpenGL or OpenGL ES. You've seen throughout the book how AV Foundation uses Core Animation to provide hardware-accelerated video rendering using the `AVPlayerLayer` and `AVVideoCapturePreviewLayer` classes. We'll soon discuss how you can also use it to build animated content for your video playback and export scenarios.

From a high-level perspective, Core Animation consists of two types of objects:

- **Layers:** Layer objects are defined by the `CALayer` class and are used to manage an element of visual content onscreen. This content is often in the form of an image or a Bezier path, but the layer itself has visual characteristics that can be set, such as its background color, opacity, and corner radius. Layers define their own geometry, such as their bounds and position, and are composed together into layer hierarchies to build more complex interfaces. In addition to the base `CALayer` class, the framework provides a number of useful subclasses, such as `CATextLayer` for rendering textual content or `CAShapeLayer` for rendering a Bezier path, both of which can be very useful when building animated overlay effects.
- **Animations:** Animation objects are instances of the abstract `CAAnimation` class, which defines the core animation behavior common to all animation types. The framework defines a number of concrete subclasses of `CAAnimation`, with the most notable being `CABasicAnimation` and `CAKeyframeAnimation`. These classes animate state changes to individual layer properties, making it easy to build both simple and complex animations. `CABasicAnimation` enables you to build simple, single-keyframe animations, which means you animate the state of a property from one state to another over a given duration. This class is useful for performing simple animations, such as animating the size, position, or background color of a layer. `CAKeyframeAnimation` enables you to perform more advanced animations by providing you more control over the “key frames” in an animation. For instance, a keyframe animation could be used to specify the timing and pacing while animating a layer along the points of a Bezier path.

Let's look a simple example of `CALayer` and `CABasicAnimation` in action. The example places a layer with the book's cover image at the center of its parent view and animates it around the z-axis, as shown in [Figure 12.1](#).



Figure 12.1 Core Animation example

[Click here to view code image](#)

```
CALayer *parentLayer = // parent layer

UIImage *image = [UIImage imageNamed:@"lavf_cover"];

CALayer *imageLayer = [CALayer layer];
// Set the layer contents to the book cover image
imageLayer.contents = (id)image.CGImage;
imageLayer.contentsScale = [UIScreen mainScreen].scale;

// Size and position the layer
CGFloat midX = CGRectGetMidX(parentLayer.bounds);
CGFloat midY = CGRectGetMidY(parentLayer.bounds);

imageLayer.bounds = CGRectMake(0, 0, image.size.width,
image.size.height);
imageLayer.position = CGPointMake(midX, midY);

// Add the image layer as a sublayer of the parent layer
[parentLayer addSublayer:imageLayer];

// Basic animation to rotate around z-axis
CABasicAnimation *rotationAnimation =
[CABasicAnimation
animationWithKeyPath:@"transform.rotation.z"];

// Rotate 360 degrees over a three-second duration, repeat
// indefinitely
rotationAnimation.toValue = @(2 * M_PI);
rotationAnimation.duration = 3.0f;
```

```

rotationAnimation.repeatCount = HUGE_VALF;

// Add and execute animation on the image layer
[imageLayer addAnimation:rotationAnimation
forKey:@"rotateAnimation"];

```

The effects of Core Animation are difficult to see in a static code example or illustration; you'll find a project in the Chapter 12 sample code directory called **CoreAnimationExample** so you can see this code in action.

The details of using Core Animation are out of the scope of this book, but I would recommend reading Nick Lockwood's *iOS Core Animation* (2014, Boston: Addison-Wesley) for a great overview of how to effectively use the Core Animation framework.

Using Core Animation with AV Foundation

Using Core Animation to build overlay effects for your video applications is nearly identical to using it for building real-time animations for iOS or OS X. The biggest difference is in the timing model used to run your animations. When working with real-time animations, instances of CAAnimation derive their timing from the system host clock (see [Figure 12.2](#)).

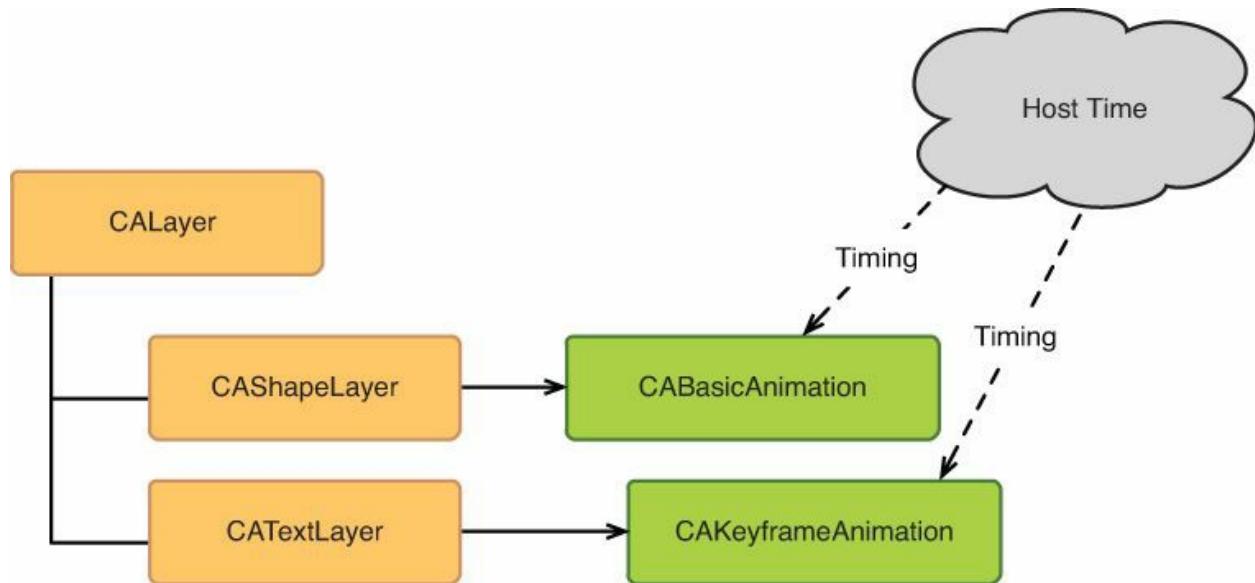


Figure 12.2 Scenario: real-time animations

Host time begins at system boot and marches forward monotonously toward infinity. Basing your animation timing on the host clock works well in the real-time case but isn't suitable for building video animations. Video animations need to operate on "movie time," which begins at the beginning

of the movie and runs to its duration. Whereas host time always moves forward, never ceasing, movie time can be stopped, paused, rewound, or fast-forwarded. For your animations to be tightly bound to the video timeline, you need to use a different timing model. AV Foundation provides two solutions for this depending on your use case. Let's begin by looking at the playback scenario.

Playback with AVSynchronizedLayer

AV Foundation provides a specialized CALayer subclass called `AVSynchronizedLayer` that synchronizes its timing with a given instance of `AVPlayerItem`. This layer isn't used to display any content itself, but simply confers its timing on to its layer subtree. This enables any animations attached to layers in its hierarchy to derive their timing from the active `AVPlayerItem` instance (see [Figure 12.3](#))

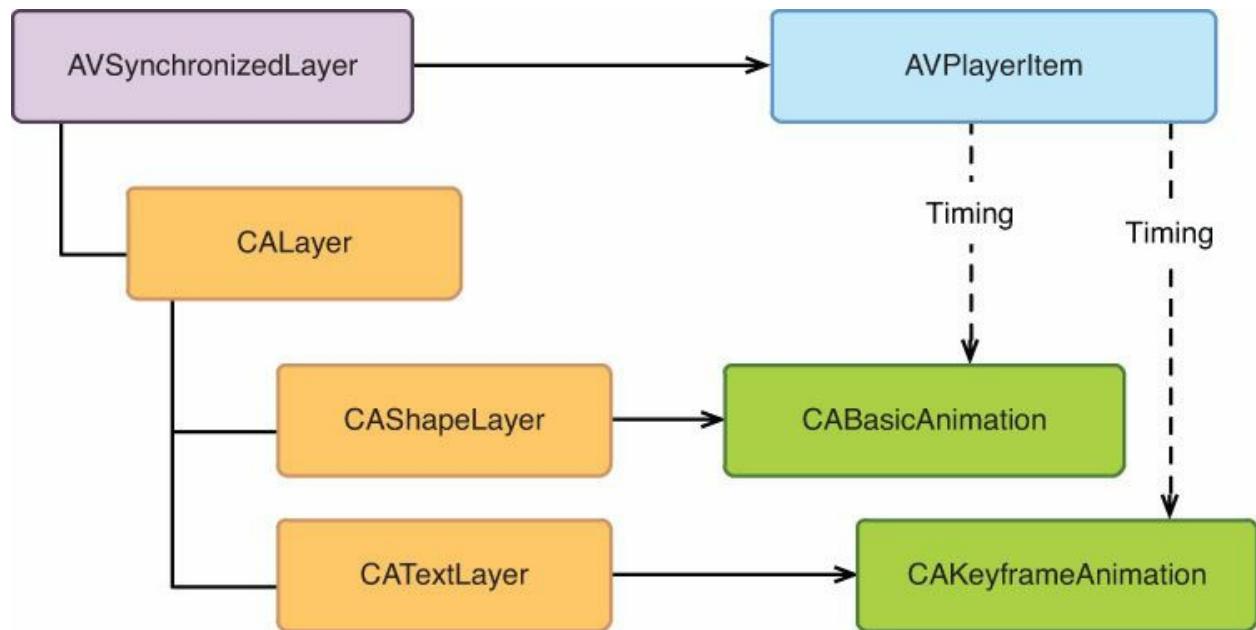


Figure 12.3 Scenario: playback animations

The way you'll typically use `AVSynchronizedLayer` is to incorporate it into the layer hierarchy of your player view, with the synchronized layer presented directly above your video layer (see [Figure 12.4](#)). This enables you to add animated titles, watermarks, or lower thirds to your video playback that stays perfectly in sync with the transport behavior of your player.



Figure 12.4 Integrating AVSynchronizedLayer

I'll let you in on a little secret. You've been using `AVSynchronizedLayer` all along. Although the primary use case for `AVSynchronizedLayer` is for animating layered content on top of your video content, it's not the only way it can be useful to you. You've seen that when you play a composition in the 15 Seconds app, an animated play head moves in sync with the playback. This is made possible thanks to `AVSynchronizedLayer`. In the project you'll find a class called `THPlayheadView` under the `Views` group. Whenever a new `AVPlayerItem` is created, the application calls this view's `synchronizeWithPlayerItem:` method. This method creates two `CAShapeLayer` instances to draw the vertical lines and an instance of `CABasicAnimation` to animate the x-position of those layers as the video plays. I won't go into the details of this view here, but I'd recommend that you look at the implementation of this class to get a better understanding of how it works.

Exporting with AVVideoCompositionCoreAnimationTool

To incorporate your Core Animation layers and animations into your exported videos, you use a class called `AVVideoCompositionCoreAnimationTool`. This class is used by an `AVVideoComposition` to incorporate your Core Animation effects as a video composition post-processing stage (see [Figure 12.5](#)).

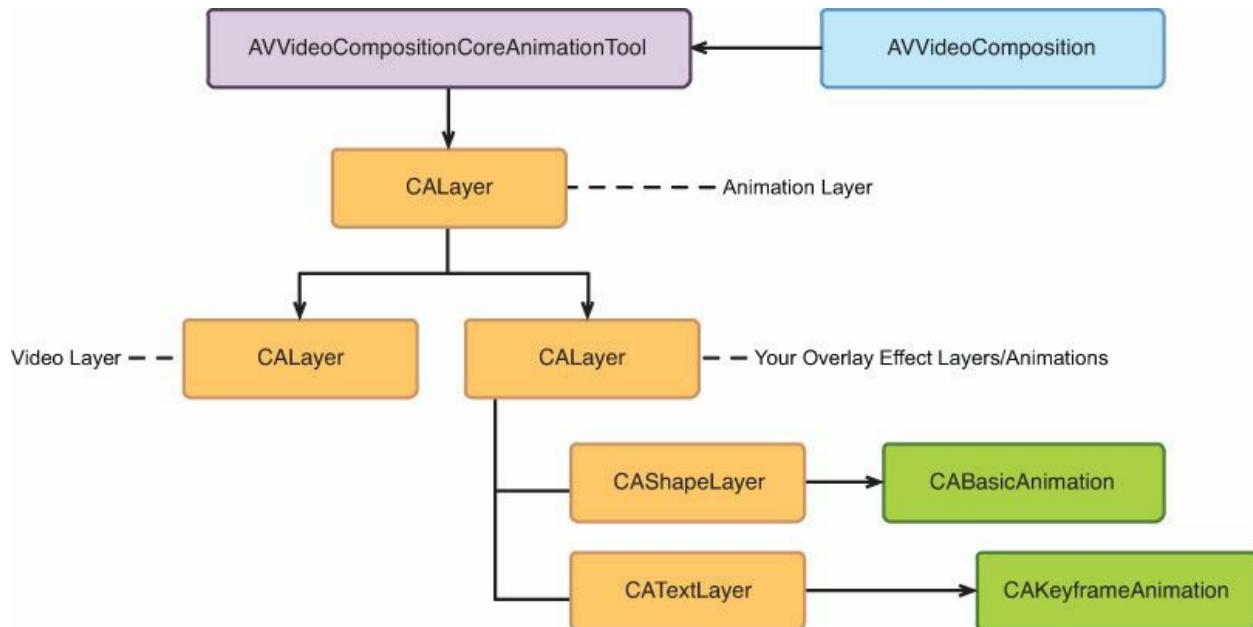


Figure 12.5 Scenario: export animations

`AVVideoCompositionCoreAnimationTool` is kind of a magical black box, but fortunately one that works well after you understand how to use it. It places the composited video frames in the video layer and then renders the animation layer containing your custom overlay effects to produce the final video frame.

The details of using `AVSynchronizedLayer` and `AVVideoCompositionCoreAnimationTool` will be discussed shortly, but first I want to mention a couple points you'll need to observe when building overlay effects:

- The Core Animation framework's default behavior is to execute an animation and dispose of it after the animation transaction completes. This is typically the behavior you want in the real-time case, because you can never go back in time. However, with video animations this would be problematic, so you need to disable this behavior by setting

an animation's `removedOnCompletion` property to NO. Failing to do so means you'll see the animation once, but it won't be seen again if you replay the video or scrub around in its timeline.

- Animations that have a `beginTime` of 0.0 will not be seen. Core Animation translates a `beginTime` of 0.0 into `CACurrentMediaTime()`, which is the current host time, which will not correspond to a valid time in the movie's timeline. If you need an animation to start at the very beginning of a movie, use the constant value `AVCoreAnimationBeginTimeAtZero` for the animation's `beginTime` property.

15 Seconds: Adding Animated Titles

One of the challenges of using Core Animation with `AVComposition` is reconciling the different conceptual and time models used. When working with `AVComposition`, you're thinking in terms of tracks and `CMTime` and `CMTimeRange` values. Core Animation has no notion of tracks and uses floating-point values to specify time. In a simple scenario you can deal with Core Animation in its terms, but when you're building more elaborate cases, it's better to provide a common level of abstraction over each framework to provide a normalized model for building your compositions and animations. Let's take a look at how this is being done in the 15 Seconds app.

You were first introduced to the application's data model in [Chapter 9](#), "[Composing and Editing Media](#)." Let's take another look at it and see if this can be extended to incorporate your Core Animation work (see [Figure 12.6](#)).

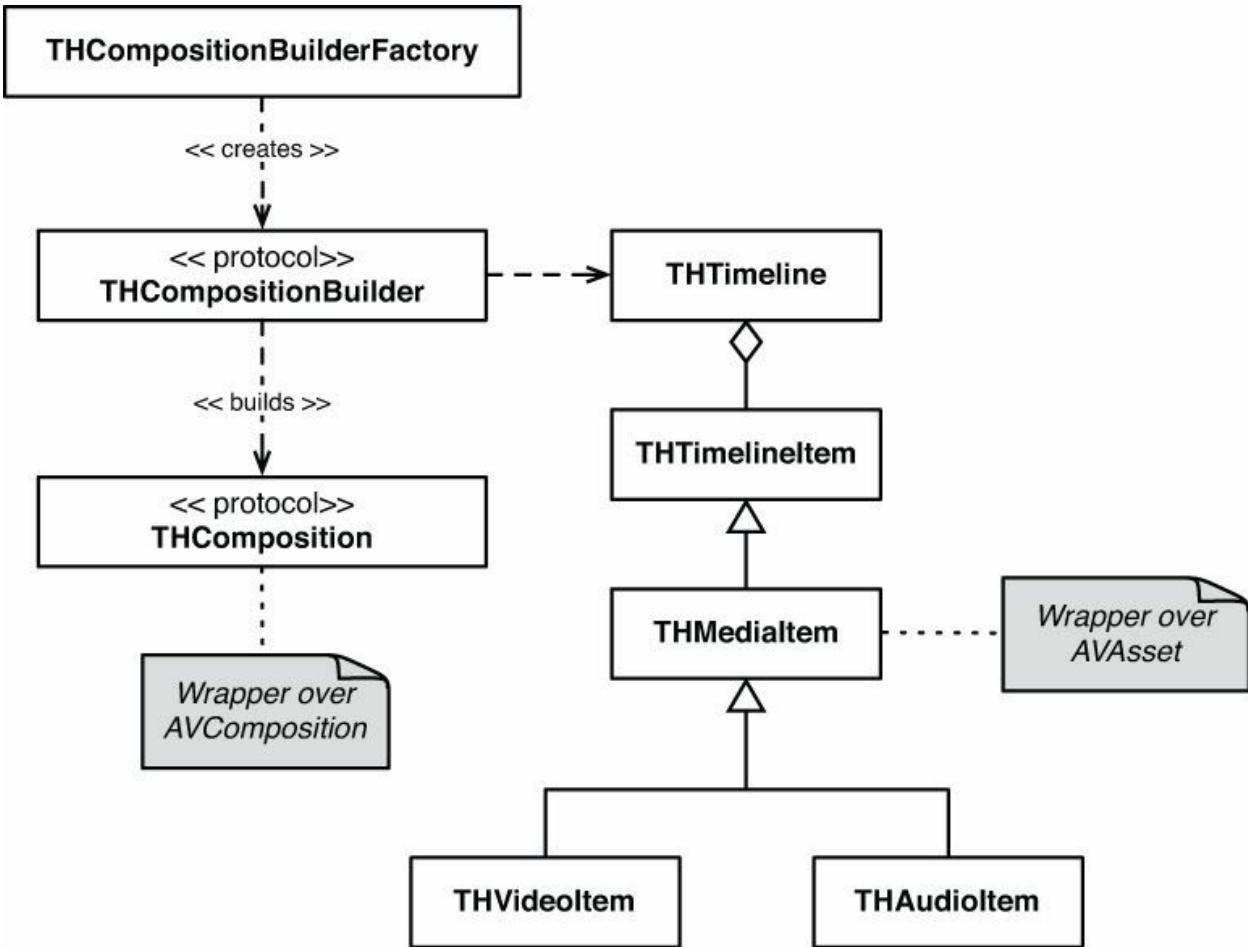


Figure 12.6 15 Seconds data model

All the elements that are represented in the application's timeline area extend from a common base class called `THTimelineItem`. This class defines the item's time range and also its starting position within the timeline.

`THMediaItem` and its subclasses `THVideoItem` and `THAudioItem` are subclasses of this base class and are used when working with the project's audio and video assets. It would be nice to treat your overlay animations in the same way so they can be visually depicted and arranged in the application's timeline view. This can be accomplished by implementing the application's animation behavior in a subclass of `THTimelineItem`.

Wrapping your animation code this way will enable it to be rendered as an item in the timeline view and also provides a common interface for working with your animation code when building your composition.

The application keeps it pretty simple from a Core Animation perspective, so you won't have to be a Core Animation expert to follow along. You'll build a

simple THTimelineItem object called THTitleItem that will be used to add animated titles to the project. You'll find a stubbed version of this class in this chapter's starter project under the Models group. Let's take a look at its interface in [Listing 12.1](#).

Listing 12.1 **THTitleItem** Interface

[Click here to view code image](#)

```
#import "THTimelineItem.h"
#import <QuartzCore/QuartzCore.h>

@interface THTitleItem : THTimelineItem

+ (instancetype)titleItemWithText:(NSString *)text image:(UIImage *)
    *image;
- (instancetype)initWithText:(NSString *)text image:(UIImage *)
    *image;

@property (copy, nonatomic) NSString *identifier;
@property (nonatomic) BOOL animateImage;
@property (nonatomic) BOOL useLargeFont;

- (CALayer *)buildLayer;

@end
```

This object is initialized with a text string and an image composed of the title elements to be displayed, and it provides properties to customize its presentation. The key method this object provides is the buildLayer method that creates the Core Animation layers and animations. Let's begin to build out this functionality starting by building the Core Animation layers as shown in [Listing 12.2](#).

Listing 12.2 **THTitleItem** Building Layers

[Click here to view code image](#)

```
#import "THTitleItem.h"
#import "THConstants.h"

@interface THTitleItem ()
@property (copy, nonatomic) NSString *text;
@property (strong, nonatomic) UIImage *image;
@property (nonatomic) CGRect bounds;
```

```

@end

@implementation THTitleItem

+ (instancetype)titleItemWithText:(NSString *)text image:(UIImage *)
    *image {
    return [[self alloc] initWithText:text image:image];
}

- (instancetype)initWithText:(NSString *)text image:(UIImage *)
    *image {
    self = [super init];
    if (self) {
        _text = [text copy];
        _image = image;
        _bounds =
TH720pVideoRect; // 1
    }
    return self;
}

- (CALayer *)buildLayer {

    // --- Build Layers

    CALayer *parentLayer = [CALayer
layer]; // 2
    parentLayer.frame = self.bounds;
    parentLayer.opacity = 0.0f;

    CALayer *imageLayer = [self makeImageLayer];
    [parentLayer addSublayer:imageLayer];

    CALayer *textLayer = [self makeTextLayer];
    [parentLayer addSublayer:textLayer];

    return parentLayer;
}

- (CALayer *)makeImageLayer // 3
{
    CGSize imageSize = self.image.size;

    CALayer *layer = [CALayer layer];
    layer.contents = (id) self.image.CGImage;
    layer.allowsEdgeAntialiasing = YES;
    layer.bounds = CGRectMake(0.0f, 0.0f, imageSize.width,
imageSize.height);
    layer.position = CGPointMake(CGRectGetMidX(self.bounds) -

```

```

20.0f, 270.0f);

    return layer;
}

- (CALayer *)makeTextLayer // 4
{
    CGFloat fontSize = self.useLargeFont ? 64.0f : 54.0f;
    UIFont *font = [UIFont fontWithName:@"GillSans-Bold"
size:fontSize];

    NSDictionary *attrs =
        @{@"NSFontAttributeName" : font,
         NSForegroundColorAttributeName : (id) [UIColor
whiteColor].CGColor};

    NSAttributedString *string =
        [[NSAttributedString alloc] initWithString:self.text
attributes:attrs];

    CGSize textSize = [self.text sizeWithAttributes:attrs];

    CATextLayer *layer = [CATextLayer layer];
    layer.string = string;
    layer.bounds = CGRectMake(0.0f, 0.0f, textSize.width,
textSize.height);
    layer.position = CGPointMake(CGRectGetMidX(self.bounds),
470.0f);
    layer.backgroundColor = [UIColor clearColor].CGColor;

    return layer;
}

@end

```

- 1.** You define a bounds variable initialized with the value TH720pVideoRect. This is a CGRect constant value found in THConstants.h defining a rectangle with an origin of 0, 0 and a size of 1280, 720. This matches the dimensions of the application's videos.
- 2.** The next step is to build the Core Animation layer objects. You create a parent layer to contain the image and text layers and size it appropriately. You also want to set its initial opacity to 0.0 so it will not be visible. You'll later animate this property to control its visibility.

3. Create a new CALayer to display the image content, and set its contents property to the CGImageRef representation of the title item's image. Core Animation deals only with Core Graphics types, so you'll need to ask the UIImage for its underlying CGImageRef. Set the layer's allowsEdgeAntialiasing property to YES so the edges of the image will be anti-aliased when the image is animated. Finally, you set the layer's bounds and position as needed and return the layer.
4. Create a new CATextLayer to render the string content. The easiest way of working with a CATextLayer is to use an NSAttributedString, so you'll build the needed NSAttributedString attributes dictionary containing the desired font and foreground color values and create an NSAttributedString for the title item's text. Set the attributed string as the layer's string property, adjust its bounds and position as shown, and return the layer.

With the layers created you have the structural parts of the title overlay complete, but if you attempted to use this in its current state nothing would be shown because the parent layer's opacity is set to 0.0. This is a good starting state, but you need to add some animations to the mix before this can be put to use. Let's add the needed animations starting with the fade in/fade out animation shown in [Listing 12.3](#).

Listing 12.3 Adding the Fade In and Out Animation

[Click here to view code image](#)

```
@implementation THTitleItem

...
- (CALayer *)buildLayer {
    // --- Build Layers
    CALayer *parentLayer = [CALayer layer];
    parentLayer.frame = self.bounds;
    parentLayer.opacity = 0.0f;

    CALayer *imageLayer = [self makeImageLayer];
    [parentLayer addSublayer:imageLayer];
```

```

CALayer *textLayer = [self makeTextLayer];
[parentLayer addSublayer:textLayer];

// --- Build and Attach Animations

CAAnimation *fadeInFadeOutAnimation = [self
makeFadeInFadeOutAnimation];
[parentLayer addAnimation:fadeInFadeOutAnimation
forKey:nil]; // 1

return parentLayer;
}

...
- (CAAnimation *)makeFadeInFadeOutAnimation {

CAKeyframeAnimation *animation =
[CAKeyframeAnimation animationWithKeyPath:@"opacity"];

animation.values = @[@0.0f, @1.0, @1.0f,
@0.0f]; // 2
animation.keyTimes = @[@0.0f, @0.20f, @0.80f, @1.0f];

animation.beginTime =
CMTimeGetSeconds(self.startTimeInTimeline); // 3
animation.duration =
CMTimeGetSeconds(self.timeRange.duration);

animation.removedOnCompletion =
NO; // 4

return animation;
}

@end

```

1. The parent layer's opacity is set to 0.0, so you'll want to create an animation that will fade the layer's opacity in and out over the appropriate time range. You create this animation by calling `makeFadeInFadeOutAnimation` and adding it to the parent layer using the `addAnimation:forKey:` method. You can specify an `NSString` for the `forKey:` argument if you need to later identify or retrieve this animation, but in this case that won't be necessary, so you'll just pass a `nil` key value for this argument.

2. The behavior you want for each title is to quickly fade in, hold at its full opacity for a given duration, and then fade out at the end. This is a good use case for a keyframe animation, so you create a new instance of `CAKeyframeAnimation` that will operate on the layer's `opacity` property. You define an array of opacity values and a corresponding array of associated `keyTimes` indicating when those opacity values should be set. The time values are always specified on a normalized scale from 0.0 to 1.0 and are made relative to the animation's duration. You could get more precise in the calculations of these times, but for the example app you'll say the first 20% of the animation will fade in and the last 20% will fade out.
3. Use the two time-related properties defined by `THTimelineItem` to determine the animation's `beginTime` and `duration`. Use the `CMTimeGetSeconds` function to convert the `startTimeInTimeline` property to a floating-point value to be set as the animation's `beginTime` property. Similarly, get the item's `timeRange.duration` value in seconds and set it as the animation's duration.
4. You want to remember to always set the animation's `removedOnCompletion` property to `NO` so the animation doesn't get removed after it executes. Failing to explicitly set this property means you'll see the animation once, but never again.

This takes care of the core fade in and out behavior you want all title items to have. To make things a little more interesting, it would be nice to also have the capability of animating the title item's image. Specifically, you'll apply a spin and pop animation to the image if the title item's `animateImage` property is set to `YES`. Let's look at how to implement the methods to create those animation effects in [Listing 12.4](#).

Listing 12.4 Animating the Title Image

[Click here to view code image](#)

```
- (CALayer *)buildLayer {  
    // --- Build Layers  
  
    CALayer *parentLayer = [CALayer layer];  
    parentLayer.frame = self.bounds;
```

```

parentLayer.opacity = 0.0f;

CALayer *imageLayer = [self makeImageLayer];
[parentLayer addSublayer:imageLayer];

CALayer *textLayer = [self makeTextLayer];
[parentLayer addSublayer:textLayer];

// --- Build and Attach Animations

CAAnimation *fadeInFadeOutAnimation = [self
makeFadeInFadeOutAnimation];
[parentLayer addAnimation:fadeInFadeOutAnimation forKey:nil];

if (self.animateImage) {

    parentLayer.sublayerTransform =
THMakePerspectiveTransform(1000); // 1

    CAAnimation *spinAnimation = [self make3DSpinAnimation];

    NSTimeInterval offset
= // 2
      spinAnimation.beginTime + spinAnimation.duration -
0.5f;

    CAAnimation *popAnimation =
      [self makePopAnimationWithTimingOffset:offset];

    [imageLayer addAnimation:spinAnimation
forKey:nil]; // 3
      [imageLayer addAnimation:popAnimation forKey:nil];

}

return parentLayer;
}

...

static CATransform3D THMakePerspectiveTransform(CGFloat
eyePosition) {
    CATransform3D transform = CATransform3DIdentity;
    transform.m34 = -1.0 / eyePosition;
    return transform;
}

- (CAAnimation *)make3DSpinAnimation {

```

```
// To be implemented

    return nil;
}

- (CAAnimation *)makePopAnimationWithTimingOffset:
(NSTimeInterval)offset {

    // To be implemented

    return nil;
}

@end
```

1. You'll be applying a 3D spin animation around the y-axis. To achieve this effect, you need to set a perspective transform as the parent layer's `sublayerTransform` property. You used this same effect when we discussed Face Detection in [Chapter 7, “Using Advanced Capture Features.”](#) This gives the effect of the image's 2D plane being projected into a 3D space.
 2. Create a `CAAnimation` to apply the spin animation and then calculate an offset value, which will be used to build the pop animation. The offset will take the spin animation's `beginTime` and `duration` and then subtract half a second from that value. This has the effect of applying the pop animation to the last half-second of the spin.
 3. Add both animations to the `imageLayer` using the `addAnimation:forKey:` method.
-

Note

Core Animation provides a class called `CAAnimationGroup`, which is used to group multiple animations together to be executed concurrently. If you're an experienced Core Animation developer, you may be wondering why I'm not grouping the spin and pop animations. Although I've never gotten a definitive answer that animation groups don't work when using them with `AVSynchronizedLayer`, I personally have not had any luck using them when building video animations.

Let's finish off the `THTitleItem` class by providing implementations of the two outstanding animation methods shown in [Listing 12.5](#).

Listing 12.5 Image Animation Methods

[Click here to view code image](#)

```
- (CAAnimation *)make3DSpinAnimation {

    CABasicAnimation *animation
    = [CABasicAnimation
        animationWithKeyPath:@"transform.rotation.y"];
        // 1

    animation.toValue = @((4 * M_PI) *
    -1); // 2

    animation.beginTime =
    CMTimeGetSeconds(self.startTimeInTimeline) + 0.2; // 3
    animation.duration = CMTimeGetSeconds(self.timeRange.duration)
    * 0.4;

    animation.removedOnCompletion = NO;

    animation.timingFunction
    = [CAMediaTimingFunction
        functionWithName:kCAMediaTimingFunctionEaseInEaseOut];
        // 4

    return animation;
}

- (CAAnimation *)makePopAnimationWithTimingOffset:
(NSTimeInterval)offset {

    CABasicAnimation *animation
    = [CABasicAnimation
        animationWithKeyPath:@"transform.scale"];
        // 5

    animation.toValue =
    @1.3f; // 6

    animation.beginTime =
    offset; // 7
    animation.duration = 0.35f;

    animation.autoreverses =
    YES; // 8
```

```
        animation.removedOnCompletion = NO;

        animation.timingFunction =
            [CAMediaTimingFunction
                functionWithName:kCAMediaTimingFunctionEaseInEaseOut];

        return animation;
    }
}
```

1. You begin by creating a `CABasicAnimation` to animate the image's transform. The Core Animation Programming Guide details some of the Key-Value Coding Extensions it provides that make this easy to do. In this case you'll use the `transform.rotation.y` key path to rotate the image around the y-axis.
2. The effect to apply is to rotate the image 2 times around the y-axis in a counterclockwise manner. A full 360° rotation is 2π radians so you'll use $4 * M_PI$ to achieve two full rotations. You multiply that value by -1 to have the rotation move in a counterclockwise manner.
3. You calculate the `beginTime` and `duration` similar to the fade in/out animation. You need to make some small calculations to adjust the timing of these. There's no magic math to these numbers, just what looked good to me, so feel free to adjust these as you please.
4. Apply a timing function to this animation. By default an animation uses a linear timing curve when calculating its interpolated values. This generally feels a bit stiff, and it's often preferable to apply some "easing" to that curve to feel a bit more natural. In this case you'll use an *ease in/ease out* curve.
5. For the pop animation you'll create a `CABasicAnimation` to manipulate the image's transform to apply a small scaling effect.
6. Set the animation's `toValue` to `@1.3` to scale the image up by 30%.
7. Because this animation will run as part of the previous animation, set the `beginTime` to the offset value passed to the method, and set the duration to 0.35. The specific `duration` value is somewhat arbitrary, so feel free to set it according to your preference.
8. Animation objects have the capability of autoreversing their animations. This is a handy feature that will execute the animation and

then run the animation in reverse, returning it to its initial state. Setting the autoreverses property to YES quickly scales the image up and down to produce a nice little “pop” effect at the end of the animation.

The THTitleItem class is complete, but you’ve got a bit of work to do before you can see it in action. I’d recommend running this chapter’s **FifteenSeconds_Final** project to see these animations in action. A couple new user interface elements have been added to the app, enabling you to experiment with the animations. The Settings menu now contains a Video Titles switch to enable the animation track. A UISlider has also been added to the transport area so you can scrub around as you’re playing the composition, and you can see how the animations stay perfectly locked to the timeline (see [Figure 12.7](#)).

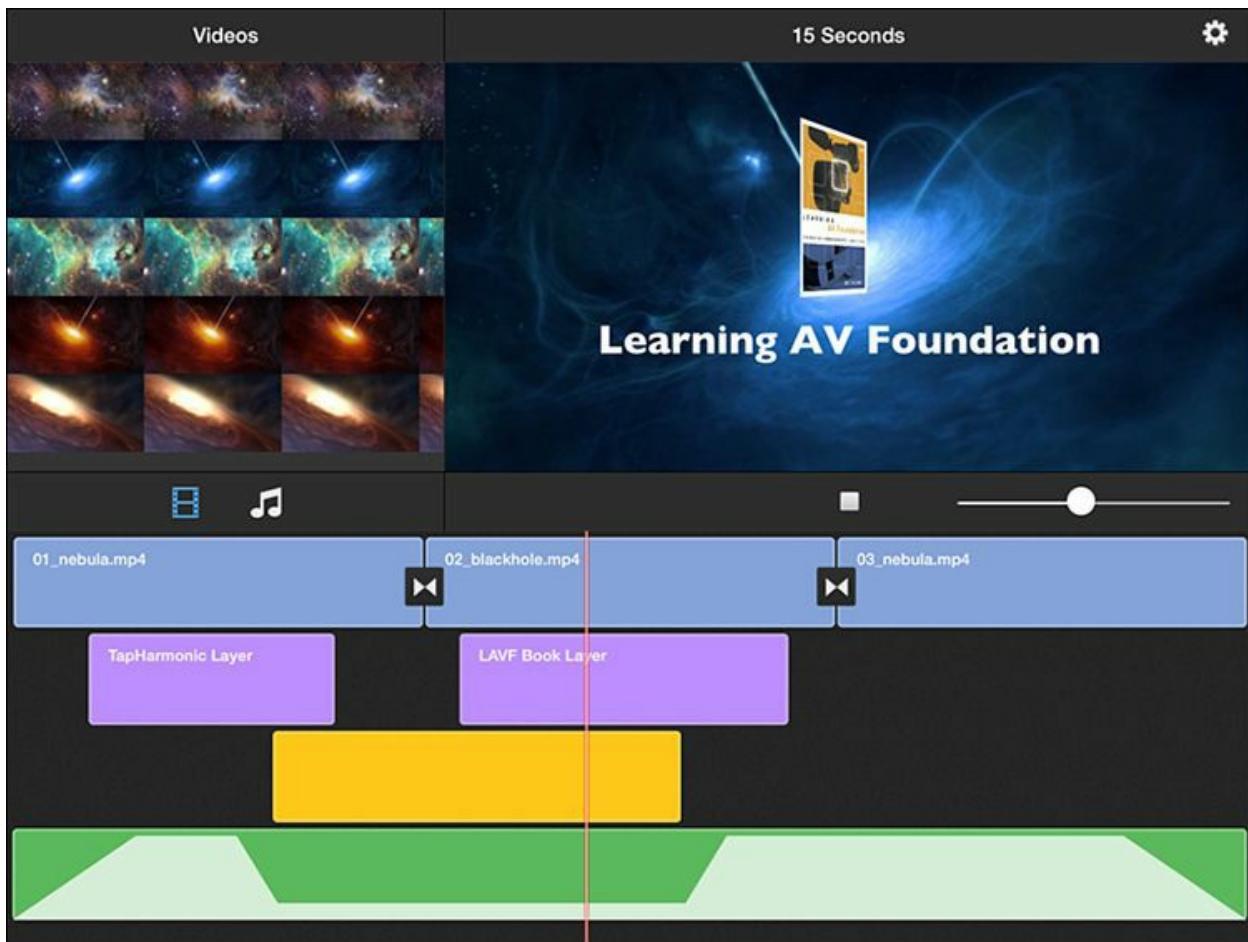


Figure 12.7 Animations in action

Note

When you enable the Video Titles switch in the menu, this adds two

pre-canned titles to the timeline area. The code that constructs these title items can be found in the `addTitleItems` method of the `THTimelineViewController` class.

In the next section, you learn how to make use of the `THTitleItem` in the your compositions.

Preparing the Composition

You won't need to create new `THComposition` and `THCompositionBuilder` instances in this chapter because the ones you created in the previous chapter already do most of what you need. You'll find the classes `THOverlayComposition` and `THOverlayCompositionBuilder` in the sample project that are copies of the work you did in the previous chapter, but with a few small changes made to incorporate the Core Animation work you created in the previous section (see [Listing 12.6](#)).

Listing 12.6 `THOverlayComposition` Interface

[Click here to view code image](#)

```
@interface THOverlayComposition : NSObject <THComposition>

@property (strong, nonatomic, readonly) AVComposition
*composition;
@property (strong, nonatomic, readonly) AVVideoComposition
*videoComposition;
@property (strong, nonatomic, readonly) AVAudioMix *audioMix;
@property (strong, nonatomic, readonly) CALayer *titleLayer;

- (id)initWithComposition:(AVComposition *)composition
                  videoComposition:(AVVideoComposition *)videoComposition
                           audioMix:(AVAudioMix *)audioMix
                         titleLayer:(CALayer *)titleLayer;
@end
```

This is the same code you created in the previous chapter, but with the addition of a new `CALayer` property and a new `titleLayer` argument in the initializer. Before we discuss the changes you'll need to make to the class implementation to use the `THTitleItem` content, let's first discuss the additions to make to the `THOverlayCompositionBuilder` class.

The composition builder requires some fairly minor changes. The key thing is to take the `THTitleItem` instances in the `THTimeline`, aggregate them into a single `CALayer` object, and add them to the `THOverlayComposition`. [Listing 12.7](#) provides the details of these changes.

Listing 12.7 Building the Video Layers

[Click here to view code image](#)

```
@implementation THOverlayCompositionBuilder

...
- (id <THComposition>)buildComposition {
    self.composition = [AVMutableComposition composition];
    [self buildCompositionTracks];
    AVVideoComposition *videoComposition = [self
        buildVideoComposition];
    return [[THOverlayComposition
alloc] // 1
        initWithComposition:self.composition
        videoComposition:videoComposition
        audioMix:[self buildAudioMix]
        titleLayer:[self buildTitleLayer]];
}
- (CALayer *)buildTitleLayer {
    if (!THTitleIsEmpty(self.timeline.titles)) {
        CALayer *titleLayer = [CALayer
layer]; // 2
        titleLayer.bounds = TH720pVideoRect;
        titleLayer.position
= CGPointMake(CGRectGetMidX(TH720pVideoRect),
        CGRectGetMidY(TH720pVi
            for (THTitleItem *compositionLayer in
self.timeline.titles) { // 3
                [titleLayer addSublayer:[compositionLayer
buildLayer]];
            }
        return titleLayer;
    }
}
```

```
    }

    return nil;
}

...
@end
```

1. You create and return an instance of `THOverlayComposition`.
The only change from the previous chapter is that you call the private `buildTitleLayer` method to return a `CALayer` instance containing your animated layer effects.
2. Create a new `CALayer` instance and set its bounds and position.
You again use the `TH720pVideoRect` constant in the calculation of these values. The 15 Seconds app is using only 720p video clips, but in your own apps this layer should be sized according to your video dimensions.
3. Iterate through all the titles contained in the timeline, and for each you'll call its `buildLayer` method and add the returned layer to the `titleLayer`.

The `THOverlayCompositionBuilder` class is complete, so the last thing to discuss is how to use these layers when building the `AVPlayerItem` and `AVAssetExportSession` inside the `THOverlayComposition` class.

Using Core Animation: Playback

To incorporate these layers into the playback functionality of the app, you use the `AVSynchronizedLayer` class. The layer you created in the composition builder will be added to the synchronized layer, ensuring that your animations play in sync with the video timeline. Let's look at the `makePlayable` method in `THOverlayComposition` (see [Listing 12.8](#)).

Listing 12.8 Using Core Animation in Playback

[Click here to view code image](#)

```
- (AVPlayerItem *)makePlayable {
```

```

AVPlayerItem *playerItem =
    [AVPlayerItem playerItemWithAsset:[self.composition
copy]];

playerItem.videoComposition = self.videoComposition;
playerItem.audioMix = self.audioMix;

if (self.titleLayer)
{
    // 1
    AVSynchronizedLayer *syncLayer =
        [AVSynchronizedLayer
synchronizedLayerWithPlayerItem:playerItem];

    [syncLayer addSublayer:self.titleLayer];

    // WARNING: The the 'titleLayer' property is NOT part of
    // Provided by AVPlayerItem+THAdditions category.
    playerItem.syncLayer =
    syncLayer;                                // 2
}

return playerItem;
}

```

- 1.** If a valid `titleLayer` exists, you create a new instance of `AVSynchronizedLayer` with the current `AVPlayerItem`. Add the `titleLayer` as a sublayer, ensuring all its animations are synchronized with the `AVPlayerItem` instance.
- 2.** I have added a category method to `AVPlayerItem` to carry the `AVSynchronizedLayer` to the application's `THPlayerViewController`. This is being done only to keep our usage of `AVSynchronizedLayer` limited to this class in order to simplify the discussion. In a real application, the creation and management of `AVSynchronizedLayer` would likely be better handled in the `THPlayerViewController` itself.

Load the default composition, enable the Video Titles switch from the Settings menu, and play the composition. You now have nicely animated titles that play in sync with the video content. While the composition is playing, you can use the slider next to the Play button to scrub through the video to see how your video effects stay perfectly in sync wherever you are in the timeline. The playback scenario looks good, so let's move on and

discuss how to apply these same effects when you export your compositions.

Using Core Animation: Export

To incorporate your Core Animation layers in export scenarios, you need to make use of `AVVideoCompositionCoreAnimationTool`. This object enables you to incorporate your Core Animation layers as a post-processing stage of your video composition. This object is relatively easy to use, but not particularly intuitive, so let's see how to use this tool (see [Listing 12.9](#)).

Listing 12.9 Using Core Animation in Export

[Click here to view code image](#)

```
- (AVAssetExportSession *)makeExportable {

    if (self.titleLayer) {

        CALayer *animationLayer = [CALayer
layer];                                // 1
        animationLayer.frame = TH720pVideoRect;

        CALayer *videoLayer = [CALayer layer];
        videoLayer.frame = TH720pVideoRect;

        [animationLayer addSublayer:videoLayer];
        [animationLayer addSublayer:self.titleLayer];

        animationLayer.geometryFlipped =
YES;                                         // 2

        AVVideoCompositionCoreAnimationTool *animationTool
=                               //
3      [AVVideoCompositionCoreAnimationTool videoCompositionCoreAnimat:

        AVMutableVideoComposition *mvc =
(AVMutableVideoComposition *) self.videoComposition;

        mvc.animationTool =
animationTool;                                // 4
    }

    NSString *presetName = AVAssetExportPresetHighestQuality;
    AVAssetExportSession *session =
[[AVAssetExportSession alloc] initWithAsset:
[self.composition copy]
presetName:presetName];
    session.audioMix = self.audioMix;
```

```
    session.videoComposition = self.videoComposition;

    return session;
}
```

1. You begin by creating two instances of `CALayer`. The composited video frames will be placed in the `videoLayer` and the `animationLayer` will be rendered to produce the final video frame to be exported.
2. You need to set the animation layer's `geometryFlipped` property to YES to ensure the titles are rendered correctly. Failing to set this value will result in the positions of the image and text being inverted.
3. Create a new instance of `AVVideoCompositionCoreAnimationTool` using its `videoCompositionCoreAnimationToolWithPostProcess` initializer, passing it the animation and video layers. This renders the video frames and Core Animation layers together in a post-processing stage of the video composition.
4. You set `AVVideoCompositionCoreAnimationTool` as the video composition's `animationTool` property so it can be used upon export.

On a device, run the application and load the default composition. Be sure to enable Video Titles from the Settings menu and then tap the Export Composition menu item. After a brief export process you can switch to the iOS Photos app and play your video. You'll now see your animated title layers rendered into the exported video content, as shown in [Figure 12.8](#).

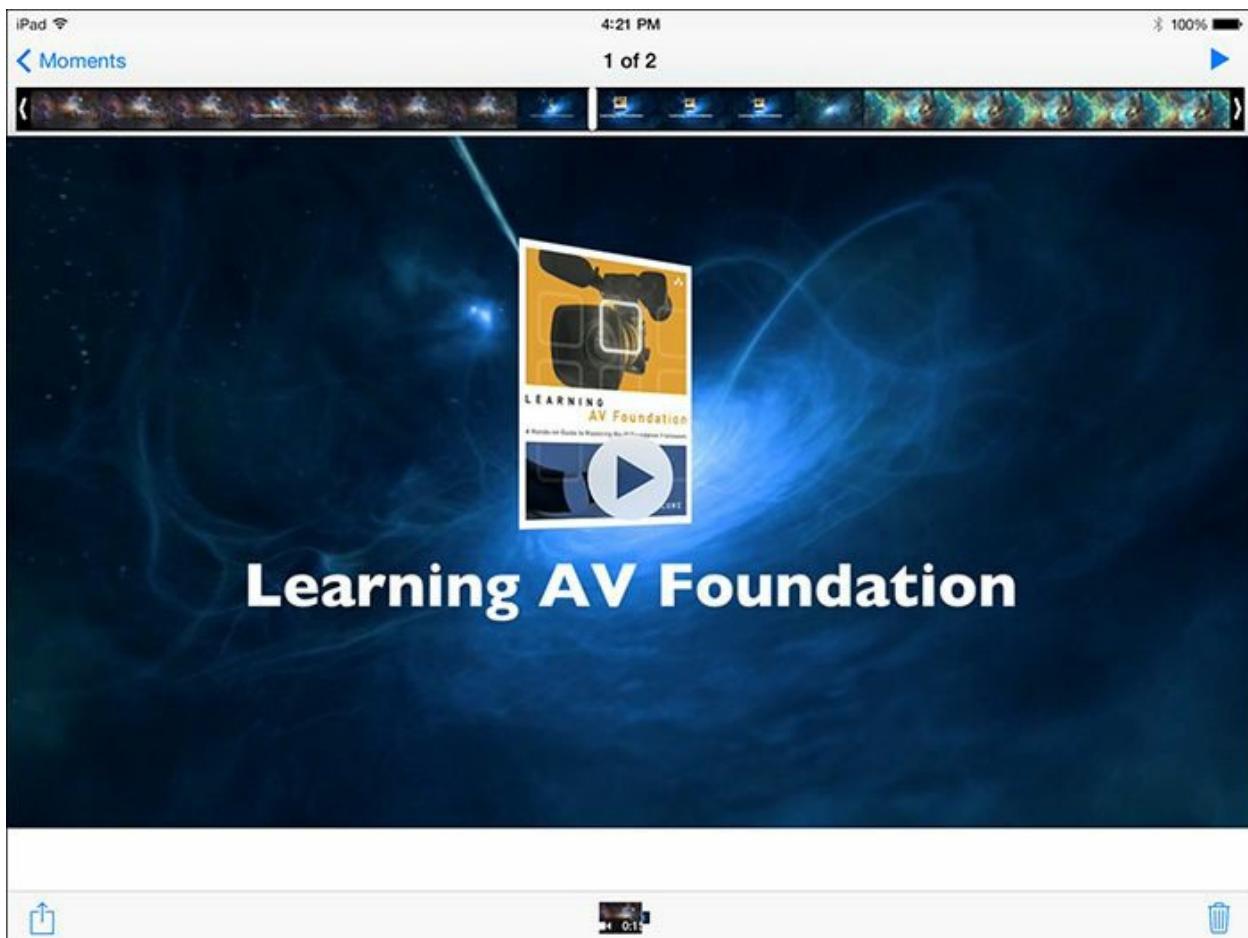


Figure 12.8 iOS Photos app with exported video

Summary

In this chapter you put the finishing touches on the 15 Seconds app and the results look great! You learned to use `AVSynchronizedLayer` to include your Core Animation effects in your video playback and saw how to use `AVVideoCompositionCoreAnimationTool` to incorporate these effects in your export process. AV Foundation's approach to adding titles to your video projects can seem a bit strange at first, but after you get comfortable with this approach it works quite well. This chapter also outlined one solution for simplifying the integration of AV Foundation and Core Animation. I believe following a similar design in your own applications can be helpful when you're building larger or more complex video applications.

Challenge

You have the Core Animation infrastructure in place for both playback and

export, so use it to come up with your own custom animations. Add a static watermark to the video. Incorporate some alternate title effects. Create the Star Wars scrolling text effect. Core Animation is a lot of fun to use, and you can use it to build beautiful animated overlay effects for your video projects.

Index

Symbols

1D barcodes, [229-230](#)

2D barcodes, [230-231](#)

15 Seconds app

 animated titles, [367-378](#)

 data model, [368](#)

 fade in/fade out animation, [372-373](#)

 image animation methods, [375-378](#)

 THTitleItem, [369-372](#)

 title image animation, [373-375](#)

 mixing audio, [327-333](#)

 buildAudioMixWithTrack: method, [331-332](#)

 Settings menu—audio controls, [333](#)

 THAudioMixComposition, [327-328](#)

 THCompositionBuilder, [328-331](#)

 THVolumeAutomation, [331](#)

 transition effects, [357](#)

 video transitions, [337](#)

 AVVideoComposition, [336](#)

 AVVideoCompositionLayerInstruction, [337](#)

 buildCompositionTracks method, [351-353](#)

 buildVideoComposition: method, [353-355](#)

 push transitions, [357-359](#)

 THCompositionBuilder, [349-351](#)

 THTransitionComposition, [348-349](#)

 transitionInstructionsInVideoComposition: method, [355-356](#)

 wipe transitions, [359-360](#)

A

AAC (Advanced Audio Coding), [18](#)

addAnimation:forKey: method, [373](#)
addBoundaryTimeObserverForTimes method, [119](#)
Adding Export Properties (listing), [159](#)
Adding the Fade In and Out Animation (listing), [372](#)
adding time, [301](#)
Adding Zoom State Observers (listing), [214](#)
addItemEndObserverForPlayerItem method, [121](#)
addMetadataItem: Implementation (listing), [83](#)
addPeriodicTimeObserverForInterval method, [119](#)
addPlayerItemTimeObserver method, [119](#)
adjustRate Method Implementation (listing), [33](#)
Adopting AVCaptureFileOutputRecordingDelegate (listing), [206](#)
Adopting AVCaptureMetadataOutputObjectsDelegate (listing), [219](#)
Advanced Audio Coding (AAC), [18](#)
AirPlay video playback, [133-135](#)
ALAsset, [61](#)
ALAssetOrientation, [202](#)
ALAssetRepresentation, [61](#)
ALAssetsLibrary, [199](#)
alwaysDiscardsLateVideoFrames, [280](#)
Ambient audio session category, [26](#)
amplitude, [8, 10](#)
analog versus digital, [7](#)
analog-to-digital conversion. *See* [sampling](#)
animated titles
 data model, [368](#)
 fade in/fade out animation, [372-373](#)
 image animation methods, [375-378](#)
 THOverlayComposition interface, [378](#)
 THTitleItem, [369-372](#)
 title image animation, [373-375](#)
animation, [361](#)
 animated titles, [367-378](#)

data model, [368](#)
fade in/fade out animation, [372-373](#)
image animation methods, [375-378](#)
THOverlayComposition interface, [378](#)
THTitleItem, [369-372](#)
title image animation, [373-375](#)

Core Animation, [5, 105](#)
animation objects, [362](#)
AVVideoCompositionCoreAnimationTool, [366-367](#)
in export, [381-383](#)
keyframe animation, [362](#)
layer objects, [362](#)
overview, [361-363](#)
playback, [364-366, 380-381](#)
timing model, [363-364](#)

preparing composition
building video layers, [379](#)
THOverlayComposition, [378](#)

animation objects, [362](#)

Apple ProRes, [17-18](#)

Applying a Dissolve Transition (listing), [357](#)

artwork conversion (MetaManager app project), [86-87](#)

aspect ratio, [12](#)

assets, [59-60](#)
asynchronous loading, [63-65](#)
building compositions, [303](#)
creating, [60-63](#)
iOS AssetsLibrary framework, [61](#)
iOS iPod Library, [62](#)
Mac OS X iTunesLibrary framework, [62-63](#)
queue management, [104](#)
tracks, [60](#)

AssetsLibrary framework, [61, 199-202](#)

asynchronous loading of asset properties, [63-65](#)

Atom Inspector, [66](#)

atoms (QuickTime), [66](#)

audio. *See also* [media](#)

capturing. *See* [capturing media](#)

Core Audio, [5](#)

looping, [29](#), [30-34](#)

configuring audio sessions, [34-36](#)

handling interruptions, [36-42](#)

responding to route changes, [40-42](#)

Mac OS X versus iOS environments, [25-26](#)

metering, [29](#), [52-57](#)

mixing

15 Seconds app, [327-333](#)

automated volume changes, [324-327](#)

AVAudioMix, [324](#)

AVAudioMixInputParameters, [324](#)

AVMutableAudioMixInputParameters, [325-326](#)

buildAudioMixWithTrack: method, [331-332](#)

overview, [323-324](#)

Settings menu—audio controls, [333](#)

THAudioMixComposition, [327-328](#)

THCompositionBuilder, [328-331](#)

THVolumeAutomation, [331](#)

playback, [6](#), [28-30](#)

recording, [6](#), [42-45](#)

sampling, [8-13](#)

storage requirements, [12](#)

timescales, [301](#)

audio channels, [44](#)

audio codecs, [18](#)

Audio Ducking switch, [333](#)

audio format (AVAudioRecorder), [43-44](#)

Audio Processing audio session category, 27

Audio Queue Services, 28

audio samples

reading, [266-270](#)

readAudioSamplesFromAsset:, [268-270](#)

THSampleDataProvider, [267](#)

reducing, [271-273](#)

rendering, [273](#)

drawRect: method, [275-276](#)

setAsset: method, [274](#)

THWaveformView, [273](#)

audio sessions, 26-28

categories, [26-27](#)

configuring, [27-28](#), [34-36](#), [46-52](#)

notifications, [37](#)

audio waveform view, building

overview, [265-266](#)

reading audio samples, [266-270](#)

reducing audio samples, [271-273](#)

rendering audio samples, [273](#)

automated volume changes, 324-327

AVAsset, 59-60, 107

asynchronous loading, [63-65](#)

AVComposition compared, [299](#)

building compositions, [303](#)

creating assets, [60-63](#)

iOS AssetsLibrary framework, [61](#)

iOS iPod Library, [62](#)

Mac OS X iTunesLibrary framework, [62-63](#)

finding metadata, [72](#)

retrieving metadata, [70-72](#)

saving metadata, [98-100](#)

AVAssetExportPresetPassthrough preset, 100

AVAssetExportSession, [98-100](#), [159-161](#), [328](#)

AVAssetImageGenerator, [124-129](#)

AVAssetImageGeneratorCompletionHandler, [128](#)

AVAssetReader, [7](#)

example, [262-265](#)

explained, [260-261](#)

illustration, [260](#)

AVAssetReaderTrackOutput, [270](#)

AVAssetTrack, [60](#), [261](#), [327](#)

finding metadata, [72](#)

retrieving metadata, [70-72](#)

AVAssetWriter, [7](#), [284-287](#)

example, [262-265](#)

explained, [261-262](#)

graph, [284-287](#)

illustration, [260](#)

AVAssetWriterInput objects, [261](#)

AVAssetWriterInputGroup, [261](#)

AVAssetWriterInputPixelBufferAdaptor, [261](#), [286](#)

AVAssetWriterPixelBufferAdaptor, [289](#)

AVAsynchronousKeyValueLoading protocol, [64](#)

AVAudioMix

15 Seconds app, [327-333](#)

automated volume changes, [324-327](#)

illustration, [324](#)

AVAudioMixInputParameters, [324](#), [327](#)

AVAudioPlayer, [6](#), [28-30](#)

audio looping, [30-34](#)

audio sessions, configuring, [34-36](#)

controlling playback, [29-30](#)

creating, [28-29](#)

handling interruptions, [36-42](#)

responding to route changes, [40-42](#)

AVAudioRecorder, [6](#), [42-45](#)

controlling recording, [44](#)

creating, [43-44](#)

Voice Memo project, [45-52](#)

configuring audio sessions, [46-52](#)

enabling audio metering, [52-57](#)

implementation, [47-52](#)

AVAudioSession, [28](#), [35](#)**AVAudioSessionInterruptionType**, [38](#)**AVAudioSessionRouteChangeNotification**, [40](#)**AVCaptureAudioDataOutput**, [171](#), [248](#), [280](#)**AVCaptureConnection**, [172](#), [197](#), [209](#)**AVCaptureDevice**, [170-171](#)

cameras

configuring, [189-190](#)

switching, [186-189](#)

creating categories, [243](#)

flash and torch modes, adjusting, [195-197](#)

focus and exposure, adjusting, [190-195](#)

video zooming, [209-216](#)

AVCaptureDeviceFormat, [242](#)**AVCaptureDeviceInput**, [171](#), [183](#), [185-186](#)**AVCaptureDevice+THAdditions (listing)**, [247](#)**AVCaptureExposureMode**, [191](#)**AVCaptureExposureModeAutoExpose**, [191](#)**AVCaptureExposureModeContinuousAutoExposure**, [191](#)**AVCaptureExposureModeLocked**, [194](#)**AVCaptureFileOutputRecordingDelegate**, [205](#)**AVCaptureFocusModeAutoFocus**, [191](#)**AVCaptureFocusModeLocked**, [191](#)**AVCaptureMetadataOutput**, [216](#), [218](#)**AVCaptureMetadataOutputObjectsDelegate**, [218-219](#)**AVCaptureMovieFileOutput**, [171](#), [184](#), [202-208](#)

AVCaptureOutput, [171](#), [197](#)
AVCaptureScreenInput, [169](#)
AVCaptureSession, [7](#), [170](#)
 configuring capture sessions, [181](#)-[184](#), [187](#)
 creating capture controller, [179](#)-[181](#)
 starting/stopping capture session, [184](#)-[185](#)
AVCaptureStillImageOutput, [171](#), [183](#), [197](#)-[199](#), [209](#)
AVCaptureVideoDataOutput, [171](#), [247](#)-[248](#), [268](#)
 CubeKamera project, [252](#)-[257](#)
 sample code, [249](#)-[250](#)
AVCaptureVideoDataOutputSampleBufferDelegate, [248](#)
AVCaptureVideoOrientation, [199](#)
AVCaptureVideoPreviewLayer, [172](#)-[173](#), [176](#)-[179](#), [209](#)
AVComposition, [298](#)-[300](#). *See also* [FifteenSeconds project](#)
AVCompositionTrack, [299](#)-[300](#)
AVCompositionTrackSegment, [299](#)-[300](#), [339](#)
averagePowerForChannel method, [53](#)
AVErrorApplicationIsNotAuthorizedToUseDevice, [185](#)
AVFormatIDKey, [43](#)-[44](#)
AV Foundation
 described, [3](#)-[4](#)
 functionality in, [6](#)-[7](#)
 position in Mac OS X/iOS media environment, [4](#)-[6](#)
AVFrameRateRange, [245](#)
AV Kit, [4](#), [137](#)
 control styles, [144](#)-[147](#)
 for iOS, [137](#)-[139](#)
 KitTime Player project, [140](#)-[144](#)
 chapters, [151](#)-[157](#)
 enabling trimming, [157](#)-[159](#)
 exporting trimmed video, [159](#)-[161](#)
 metadata, [150](#)-[151](#)
 playback stack setup, [147](#)-[151](#)

for Mac OS X, [140](#)

AVLayerVideoGravityResize, [106](#), [173](#)

AVLayerVideoGravityResizeAspect, [105](#), [172](#)

AVLayerVideoGravityResizeAspectFill, [106](#), [173](#)

AVMediaSelectionGroup, [129-133](#), [261](#)

AVMediaSelectionOption, [129-133](#), [261](#)

AVMetadataFaceObject, [216](#)

AVMetadataItem, [6](#), [71](#)

artwork conversion, [86-87](#)

comment conversion, [87-88](#)

data conversion, [84-86](#)

disc data conversion, [91-93](#)

finding metadata, [72](#)

genre data conversion, [93-96](#)

MetaManager app project, [81-98](#)

retrieving key/value pairs, [73-75](#)

track data conversion, [88-91](#)

AVMetadataItem keyString Category Method (listing), [73](#)

AVMetadataMachineReadableCodeObject, [234](#)

AVMetadataObject, [216](#)

AVMutableAudioMix, [324](#), [326](#), [332](#)

AVMutableAudioMixInputParameters, [324-326](#)

AVMutableComposition, [299](#)

AVMutableCompositionTrack, [299](#)

AVMutableMetadataItem, [86](#)

AVMutableVideoComposition, [346](#)

AVMutableVideoCompositionInstruction, [346](#)

AVMutableVideoCompositionLayerInstruction, [346](#)

AVNumberOfChannelsKey, [44](#)

AVPlayer, [6](#)

AirPlay functionality, [133-135](#)

boundary time observation, [119-120](#)

periodic time observation, [118-119](#)

video playback, [104](#)

AVPlayerItem, [6](#), [107](#), [242](#), [324](#)

- item end observation, [121](#)-[122](#)
- loading properties, [116](#)
- status property, [108](#)

AVPlayerItemStatus, [108](#)

AVPlayerItemTrack, [107](#)

AVPlayerLayer, [105](#)-[106](#), [361](#)

- implementation, [111](#)
- showing subtitles, [129](#)-[133](#)

AVPlayerView, [140](#)

- control styles, [144](#)-[147](#)
- KitTime Player project, [140](#)-[144](#)
 - chapters, [151](#)-[157](#)
 - enabling trimming, [157](#)-[159](#)
 - exporting trimmed video, [159](#)-[161](#)
 - metadata, [150](#)-[151](#)
 - playback stack setup, [147](#)-[151](#)

AVPlayerViewController, [138](#)-[139](#)

AVPlayerViewControlsStyleFloating, [145](#)

AVPlayerViewControlsStyleInline, [145](#)

AVPlayerViewControlsStyleMinimal, [145](#)

AVPlayerViewControlsStyleNone, [146](#)

AVQueuePlayer, [104](#)

AVSampleRateKey, [44](#)

AVSpeechSynthesizer, [19](#)

AVSpeechUtterance, [19](#)

AVSpeechUtteranceMaximumSpeechRate, [23](#)

AVSpeechUtteranceMinimumSpeechRate, [23](#)

AVSynchronizedLayer, [364](#)-[366](#)

AVTimedMetadataGroup, [151](#)-[157](#)

AVURLAsset, [60](#), [303](#)

AVVideoCapturePreviewLayer, [361](#)

AVVideoCodecJPEG, [197](#)
AVVideoComposition, [336](#), [347](#)
 building, [346-347](#)
 configuring, [346-347](#)
AVVideoCompositionCoreAnimationTool, [366](#)-[367](#), [381](#)-[382](#)
AVVideoCompositionInstruction, [336](#)
AVVideoCompositionLayerInstruction, [337](#)
Aztec codes, [230](#)

B

barcode scanning, [228](#)-[241](#)
baseline encoding profile, [17](#)
beginConversation Method (listing), [22](#)
beginExport method (listing), [317](#)
B-frames, [17](#)
bidirectional frames, [17](#)
bit depth, [12](#)
boundary time observation, [119](#)-[120](#)
boxes (MPEG-4), [68](#)
Buck, Erik, [254](#)
buffers, processing sample buffers, [287](#)-[289](#)
buildAudioMixWithTrack: method, [331](#)-[332](#)
buildComposition method, [351](#)
buildCompositionTracks method, [351](#)-[353](#)
building
 AVVideoComposition, [346](#)-[347](#)
 composition and layer instructions, [344](#)-[346](#)
 video layers, [379](#)
Building the Audio Mix (listing), [331](#)-[332](#)
Building the Layers (code detection) (listing), [237](#)
Building the Track Contents (listing), [315](#)
Building the Video Layers (listing), [379](#)
buildVideoComposition: method, [353](#)-[355](#)

C

CAAnimation, [362](#)

CABasicAnimation, [362-363](#), [376](#)

CAF (Core Audio Format), [49](#)

CAKeyFrameAnimation, [362](#), [373](#)

CALayer, [111](#), [362-363](#)

calculations

pass-through and transition time ranges, [341-344](#)

on time, [301](#)

camera controllers

session outputs, configuring, [278-280](#)

THCameraController interface, [278](#)

camera device codecs, [18](#)

Camera Roll, writing to, [199-202](#)

Camera Setup (barcode scanning) (listing), [232](#)

Camera Support Methods (listing), [186](#)

cameras. *See also* [capturing media](#); [Kamera project](#)

configuring, [189-190](#)

iPhone as, [169](#)

switching, [186-189](#)

capture device coordinates versus screen coordinates, [178-179](#)

captureDevicePointOfInterestForPoint method, [179](#)

Capture Output Delegate (listing), [280](#)

captureOutput:didDropSampleBuffer:fromConnection method, [248](#)

captureOutput:didOutputSampleBuffer:fromConnection method, [248](#),
[280](#)

capture recording

AVAssetWriter graph, [284-287](#)

capture output delegate, [280-281](#)

overview, [276-277](#)

sample buffer processing, [287-289](#)

session outputs, configuring, [278-280](#)

stopWriting method, [289-290](#)

THCameraController interface, [278](#)
THMovieWriter
 example, [290-292](#)
 interface, [281-282](#)
 life-cycle methods, [282-285](#)
capture sessions, [170](#)
 configuring, [181-184](#), [187](#)
 creating capture controller, [179-181](#)
 starting/stopping, [184-185](#)
Capturing a Still Image (listing), [200](#)
capturing media, [7](#)
 AVCaptureConnection, [172](#)
 AVCaptureDevice, [170-171](#)
 AVCaptureDeviceInput, [171](#)
 AVCaptureOutput, [171](#)
 AVCaptureSession, [170](#)
 AVCaptureVideoPreviewLayer, [172-173](#)
 classes, [170](#)
 CMSSampleBuffer, [249-257](#)
 format descriptions, [250](#)
 metadata attachments, [251-252](#)
 sample code, [249-250](#)
 timing information, [251](#)
 CubeKamera project, [252-257](#)
 face detection, [216-228](#)
 high frame rate video, [241-247](#)
 iOS versus Mac OS X, [169](#)
 Kamera project, [175-208](#)
 adjusting flash and torch modes, [195-197](#)
 adjusting focus and exposure, [190-195](#)
 capturing still images, [197-199](#)
 capturing videos, [202-208](#)
 configuring cameras, [189-190](#)

configuring capture session, [181-184](#)
creating capture controller, [179-181](#)
creating preview view, [176-179](#)
privacy requirements, [185-186](#)
starting/stopping capture session, [184-185](#)
switching cameras, [186-189](#)
writing to Assets Library, [199-202](#)

machine-readable code detection, [228-241](#)
processing video, [247-248](#)
sample code, [174](#)
video zooming, [209-216](#)

Capturing Still Images (listing), [198](#)

CAShapeLayer, [238](#)

categories

- audio sessions, [26-27](#)
- creating on AVCaptureDevice, [243](#)

CATransform3D, [222](#)

CD-quality audio

- bit depth, [12](#)
- sampling rate, [12](#)

CGAffineTransform, [358](#)

CGContextDrawPath, [276](#)

CGMutablePathRef, [276](#)

CGPathAddLineToPoint, [276](#)

CGPathRelease, [276](#)

changing volume automatically, [324-327](#)

chapters (KitTime Player project), [151-157](#)

chaptersForAsset method (listing), [152](#)

chroma subsampling, [13-15](#)

CIDetector, [216](#)

CIFaceFeature, [216](#)

classes

- capturing media, [170](#)

composition classes, [298](#)
CMAAttachment, [251-252](#)
CMAudioFormatDescription, [250](#)
CMBLOCKBuffer, [268](#), [270](#)
CMBLOCKBufferCopyDataBytes function, [270](#)
CMBLOCKBufferGetDataLength function, [270](#)
CMFormatDescription, [250](#)
CMFormatDescriptionRef, [245](#)
CMSampleBuffer, [197](#), [249-257](#), [268](#)
 format descriptions, [250](#)
 metadata attachments, [251-252](#)
 sample code, [249-250](#)
 timing information, [251](#)
CMSampleBufferGetAudioBufferListWithRetainedBlockBuffer
function, [268](#)
CMSampleBufferGetDataBuffer function, [268](#), [270](#)
CMSampleBufferGetImageBuffer function, [268](#), [280](#)
CMSampleBufferInvalidate function, [270](#)
CMSampleBuffer objects, [261](#)
CMTIME, [5](#), [109-110](#), [300-301](#), [343](#)
CMTIMEAdd, [301](#)
CMTIMEFlags, [300](#)
CMTIMEGetSeconds function, [373](#)
CMTIMEMake function, [109-110](#), [300](#)
CMTIMERANGE, [302-303](#), [343](#)
CMTIMERANGEFromTimeToTime, [302](#)
CMTIMERANGEMake, [302](#)
CMTIMESCALE, [300](#)
CMTIMEShow function, [300](#)
CMTIMESUBTRACT, [301](#)
CMTIMEVALUE, [300](#)
CMVideoFormatDescription, [250](#)
Code 39 barcodes, [229](#)

Code 93 barcodes, [229](#)

Code 128 barcodes, [229](#)

codecs

 audio codecs, [18](#)

 converting legacy codecs, [161-165](#)

 lossless versus lossy, [15](#)

 video codecs, [15-18](#)

CodeKamera project, [231-241](#)

color models, YUV, [13](#)

comment conversion (MetaManager app project), [87-88](#)

common key space, [71](#)

composing media, [297-300](#)

 building compositions, [303-307](#)

 exporting compositions, [316-321](#)

 FifteenSeconds project, [307-310](#)

 building composition, [311-316](#)

 view controllers, [308-310](#)

compositions, [298-300](#)

 building, [303-307](#)

 classes, [298](#)

 exporting, [316-321](#)

 FifteenSeconds project, [307-310](#)

 building composition, [311-316](#)

 view controllers, [308-310](#)

 instructions, building, [344-346](#)

 preparing

 building video layers, [379](#)

 Core Animation in export, [381-383](#)

 Core Animation in playback, [380-381](#)

 THOverlayComposition interface, [378](#)

 saving, [299](#)

compression, [13-18](#)

 chroma subsampling, [13-15](#)

codecs

lossless versus lossy, [15](#)

video codecs, [15-18](#)

conceptual steps for video transitions, [337](#)

building and configuring AVVideoComposition, [346-347](#)

building composition and layer instructions, [344-346](#)

calculating pass-through and transition time ranges, [341-344](#)

defining overlapping regions, [340-341](#)

staggering video layout, [338-340](#)

concurrent dispatch queues, [119](#)

configuring

audio sessions, [27-28](#), [34-36](#), [46-52](#)

AVVideoComposition, [346-347](#)

cameras, [189-190](#)

capture sessions, [187](#)

session outputs, [278-280](#)

Configuring the Capture Output (listing), [253](#)

Configuring the Session Outputs (listing), [233](#), [278-279](#)

connections, capturing media, [172](#), [197](#)

container formats, [18-19](#)

control styles, [144-147](#)

converting

artwork, [86-87](#)

comments, [87-88](#)

data, [84-86](#)

disc data, [91-93](#)

genre data, [93-96](#)

legacy codecs, [161-165](#)

track data, [88-91](#)

coordinates, screen versus capture device, [178-179](#)

copyCGImageAtTime method, [124](#)

Core Animation, [5](#), [105](#), [222](#)

animated titles, [367-378](#)

data model, [368](#)
fade in/fade out animation, [372-373](#)
image animation methods, [375-378](#)
THTitleItem, [369-372](#)
title image animation, [373-375](#)
animation objects, [362](#)
AVVideoCompositionCoreAnimationTool, [366-367](#)
in export, [381-383](#)
keyframe animation, [362](#)
layer objects, [362](#)
overview, [361-363](#)
in playback, [380-381](#)
playback with AVSynchronizedLayer, [364-366](#)
preparing composition
 building video layers, [379](#)
 THOverlayComposition, [378](#)
 timing model, [363-364](#)

Core Audio, 5

Core Audio Format (CAF), 49
Core Image framework, 216
Core Media, 5, 109, 156, 302-303

Core Media framework

CMSampleBuffer, [249-257](#)
 format descriptions, [250](#)
 metadata attachments, [251-252](#)
 sample code, [249-250](#)
 timing information, [251](#)

CMTIME, [300-301](#)

Core Video, 5

corners Property (listing), 239
Creating OpenGL ES Textures (listing), 256
Creating the Action Menu (listing), 153
Creating the OpenGLESTextureCache (listing), 255

CubeKamera project, [252-257](#)
customizing menus, [153](#)
cuts, [335](#)
CVImageBufferRef, [268](#)
CVOpenGLESTextureCache, [254](#)
CVPixelBuffer, [249](#), [261](#), [289](#)

D

data conversion (MetaManager app project), [84-86](#)
Data Matrix codes, [231](#)
defining overlapping regions, [340-341](#)
Determining High FPS Support (listing), [244](#)
Determining the Interruption Type (listing), [38](#)
Determining the Notification Reason (listing), [41](#)
devices, capturing media, [170-171](#). *See also* [cameras](#)
 privacy requirements, [185-186](#)
digital versus analog, [7](#)
digital camera, iPhone as, [169](#). *See also* [cameras](#); [capturing media](#); [Kamera project](#)
digital media. *See also* [media](#)
 compression, [13-18](#)
 sampling, [8-13](#)
disc data conversion (MetaManager app project), [91-93](#)
dissolve transitions, [357](#)
drawRect: method, [275-276](#)
dynamic microphones, [8-9](#)
dynamic playback controls, [139](#)

E

EAGLContext, [252](#)
EAN-8 barcodes, [229](#)
EAN-13 barcodes, [229](#)
ease in/ease out curves, [376](#)
editing media, [7](#). *See also* [composing media](#)

effects, transition

- dissolve transitions, [357](#)
- push transitions, [357-359](#)
- wipe transitions, [359-360](#)

enabling

- subtitles, [133](#)
- trimming, [157-159](#)

Enabling High Frame Rate Capture (listing), [246](#)

Enabling Trimming (listing), [157](#)

Enabling Zoom Ramping (listing), [213](#)

encoding profiles, [17](#)

Exif (exchangeable image file format) tags, [251](#)

expectsMediaDataInRealTime property (AVAssetWriterInput), [262](#)

exporting

- with AVVideoCompositionCoreAnimationTool, [366-367](#)
- compositions, [316-321](#)
- Core Animation in, [381-383](#)
- trimmed video, [159-161](#)

Exporting a Composition (listing), [317](#)

exposure, adjusting, [190-195](#)

Extracting the Transition Instructions (listing), [355-356](#)

F

face detection, [216-228](#)

FaceKamera project, [216-228](#)

fade in/fade out animation, [372-373](#)

Fade In & Out toggle switch, [333](#)

FifteenSeconds project, [307-310](#)

- building composition, [311-316](#)
- exporting composition, [316-321](#)
- view controllers, [308-310](#)

file extensions, MPEG-4 media, [69](#)

file formats. See [container formats](#)

file types, audio format compatibility, [44](#)
filterChanged: method, [284](#)
filteredSamplesForSize: method, [271](#)
Final Cut Pro X, [3](#)
final versions of projects, [19](#)
Finding Chapters (listing), [154](#)
finding metadata, [72](#)
Finishing the Writing Session (listing), [289](#)
finishing writing sessions, [289-290](#)
finishWritingWithCompletionHandler: method, [265, 290](#)
Flash and Torch Methods (listing), [196](#)
flash mode, adjusting, [195-197](#)
Floating control style, [145](#)
floating-point value, time as, [109](#)
focus, adjusting, [190-195](#)
format descriptions, processing video, [250](#)
formats for metadata
 MP3, [69-70](#)
 MPEG-4, [68-69](#)
 QuickTime, [66-68](#)
format-specific metadata, [71](#)
formattedCurrentTime method, [51](#)
frameDuration property, [347](#)
frame rate, [12](#)
frames, [12](#)
frequency, [8, 10](#)
fromDestTransform, [358](#)
ftyp atom, [66](#)

G–H

generateCGImagesAsynchronouslyForTimes method, [124](#)
generateThumbnails Implementation (listing), [126](#)
generateThumbnails method invocation (listing), [125](#)

genre data conversion (MetaManager app project), [93-96](#)
GOP (Group of Pictures), [16](#)
gravity values (video), [105](#)

H.264 video codec, [15-17](#)

Handling Interruption Began (listing), [39](#)
Handling Interruption Ended (listing), [39](#)
Handling Subtitle Selection (listing), [132](#)
Handling the Export Completion (listing), [320](#)
headers in video files, [202](#)
Hello AV Foundation project, [19-23](#)
high encoding profile, [17](#)
High Frame Rate Capture Category (listing), [243](#)
high frame rate video capture, [241-247](#)
human hearing frequency range, [10](#)

I

ID3v2 tags, [70](#)
ID3v2.2 tags, [70](#)
identifiers, retrieving metadata, [75](#)
identityTransform, [359](#)
iFrame, [17](#)
I-frames, [16-17](#)
Image Animation Methods (listing), [375-376](#)
Image Generation (listing), [125](#)
images
 animation methods, [375-378](#)
 title image animation, [373-375](#)
iMovie, [3](#)
Implementing chaptersForAsset: (listing), [152](#)
Implementing previousChapter: and nextChapter: (listing), [156](#)
Implementing startExporting: (listing), [160](#)
Implementing the buildCompositionTracks Method (listing), [351-352](#)
Implementing the buildVideoComposition Method (listing), [353-354](#)

Implementing the drawRect: Method (listing), [275-276](#)

Implementing the setAsset: Method (listing), [274](#)

Implementing the setupView Method (listing), [222](#)

Implementing titleForAsset: (listing), [150](#)

Info.plist file, [36](#)

Inline control style, [145](#)

input/output

 capturing media, [171](#)

 responding to route changes, [40-42](#)

inspecting media, [6](#)

Instagram, [3](#)

instructions property

 AVMutableVideoComposition, [346](#)

 AVVideoComposition, [347](#)

interframe compression, [16](#)

Interleaved 2 of 5 barcodes, [230](#)

interleaving, [261](#)

interruptions, handling, [36-42](#)

intraframe compression, [16](#)

Invoking generateThumbnails method (listing), [125](#)

iOS

 AssetsLibrary framework, [61](#)

 audio environment, [25-26](#)

 audio looper project. *See* projects, audio looper

 audio sessions, [26-28](#)

 categories, [26-27](#)

 configuring, [27-28](#)

 notifications, [37](#)

 AV Kit framework, [137-139](#)

 capturing media, Mac OS X versus, [169](#)

 iPod Library, [62](#)

 media environment, AV Foundation position in, [4-6](#)

iOS Core Animation (Lockwood), [222, 363](#)

iPad, 3

iPhone, 3, 169

iPod, 3

iPod Library, 62

item end observation, 121-122

Item End Observation (listing), 121

ITF 14 barcodes, 230

iTunes, 3

iTunesLibrary framework, 62-63

iTunes Store, 3

J-K

Kamera project, 175-208

capture controller, creating, 179-181

capturing

 still images, 197-199

 videos, 202-208

configuring

 cameras, 189-190

 capture session, 181-184

flash and torch modes, adjusting, 195-197

focus and exposure, adjusting, 190-195

preview view, creating, 176-179

privacy requirements, 185-186

starting/stopping capture session, 184-185

switching cameras, 186-189

writing to Assets Library, 199-202

kAudioFormatLinearPCM, 270

kCMTimeZero, 265

keyboard shortcuts, video playback controls, 147

keyframe animation, 362

key frames, 17

key spaces, 71

keyString category method, [73](#)
Key-Value Observing (KVO), [52](#), [108](#)
key/value pairs, retrieving, [73-75](#), [81-84](#)
KitTime Player project, [140-144](#)
 chapters, [151-157](#)
 converting legacy codecs, [161-165](#)
 enabling trimming, [157-159](#)
 exporting trimmed video, [159-161](#)
 metadata, [150-151](#)
 playback stack setup, [147-151](#)
KVO (Key-Value Observing), [52](#), [108](#)

L

layer instructions, building, [344-346](#)
layer objects, [362](#)
layering animation, [361](#)
 animated titles, [367-378](#)
 data model, [368](#)
 fade in/fade out animation, [372-373](#)
 image animation methods, [375-378](#)
 THTitleItem, [369-372](#)
 title image animation, [373-375](#)
Core Animation
 animation objects, [362](#)
 AVVideoCompositionCoreAnimationTool, [366-367](#)
 in export, [381-383](#)
 keyframe animation, [362](#)
 layer objects, [362](#)
 overview, [361-363](#)
 in playback, [380-381](#)
 playback with AVSynchronizedLayer, [364-366](#)
 timing model, [363-364](#)
 preparing composition

building video layers, [379](#)
THOverlayComposition, [378](#)

Learning OpenGL ES for iOS (Buck), [254](#)

legacy codecs, converting, [161-165](#)

levels method, [55](#)

linear pulse-code modulation (LPCM), [10](#)

listings

Action Menu, [153](#)
addMetadataItem: Implementation, [83](#)
Animating the Title Image, [373-375](#)
Audio Mix, [331-332](#)
AVAssetWriter Graph, [284-286](#)
AVCaptureDevice+THAdditions, [247](#)
AVCaptureFileOutputRecordingDelegate, [206](#)
AVCaptureMetadataOutputObjectsDelegate, [219](#)
AVMetadataItem keyString Category Method, [73](#)
beginConversation Method, [22](#)
buildCompositionTracks Method, [351-352](#)
buildVideoComposition Method, [353-354](#)
Camera Setup (barcode scanning), [232](#)
Camera Support Methods, [186](#)
Capture Output Delegate, [280](#)
Capturing Still Images, [198, 200](#)
Capture Output Configuration, [253](#)
chaptersForAsset method implementation, [152](#)
Core Animation in Export, [381-382](#)
Core Animation in Playback, [380](#)
corners Property, [239](#)
Dissolve Transitions, [357](#)
drawRect: Method, [275-276](#)
Export Completion, [320](#)
Export Properties, [159](#)
Exporting a Composition, [317](#)

Extracting the Transition Instructions, [355-356](#)
Fade In and Out Animation, [372](#)
Finding Chapters, [154](#)
Finishing the Writing Session, [289](#)
Flash and Torch Methods, [196](#)
generateThumbnails method, [125-126](#)
Handling Interruptions, [39](#)
High FPS Support, [244](#)
High Frame Rate Capture, [243](#), [246](#)
Image Animation Methods, [375-376](#)
Image Generation, [125](#)
Interruption Notifications, [37](#)
Interruption Type, [38](#)
Item End Observation, [121](#)
Layer Building (code detection), [237](#)
loadMediaOptions, [131-132](#)
MainViewController Time Polling, [52](#)
metadataItems method, [96](#)
Monitoring the Export Progress, [319](#)
Movie Modernization Preparation, [162](#)
Notification Reason, [41](#)
observeValueForKeyPath method modification, [154](#)
OpenGL ES Textures, [256](#)
OpenGLESTextureCache, [255](#)
Periodic Time Observations, [119](#)
Playback Stack, [148](#)
prepareWithCompletionHandler: Implementation, [79](#)
previousChapter: and nextChapter:, [156](#)
Private THQualityOfService Class, [243](#)
Reading the Asset's Audio Samples, [268-270](#)
Resetting Focus and Exposure, [194](#)
Route Change Notifications, [40](#)
Running the Modernization, [163](#)

Sample Buffer Processing, [287-288](#)
Scrubbing Methods, [123](#)
Session Outputs Configuration, [233](#), [278-279](#)
setAsset: Method, [274](#)
setupSession: Method, [181](#)
setupView Method implementation, [222](#)
startExporting method implementation, [160](#)
Starting and Stopping the Capture Session, [184](#)
status Property observation, [117](#)
Stop Playback on Headphone Unplug, [42](#)
Subtitle Selection, [132](#)
Switching Cameras, [188](#)
Tap-to-Expose Methods, [192](#)
Tap-to-Focus Implementation, [190](#)
THAppDelegate Audio Session Setup, [35](#), [46](#)
THArtworkMetadataConverter Implementation, [86](#)
THAudioMixComposition, [327-328](#)
THAudioMixCompositionBuilder, [329-331](#)
THBasicComposition, [311](#)
THBasicCompositionBuilder, [313](#)
THCameraController
 implementation, [212](#), [217](#)
 interface, [180](#), [211](#), [217](#), [231](#), [252](#), [278](#)
THCommentMetadataConverter, [87](#)
THComposition, [311](#)
THCompositionBuilder, [313](#)
THCompositionExporter, [317](#)
THDefaultMetadataConverter, [85](#)
THDiscMetadataConverter, [91](#)
THDocument, [143](#)
THGenreMetadataConverter, [94](#)
THMainViewController Metering Methods, [56](#)
THMediaItem

implementation, [78](#)
interface, [77](#)
saveWithCompletionHandler: Implementation, [99](#)

THMetadata, [82](#)

THMetadataConverter, [85](#)

THMetadataItem, [81](#)

THMeterTable, [53](#)

THMovieWriter
 interface, [281](#)
 life-cycle methods, [282-284](#)
 usage, [290-292](#)

THOverlayComposition, [378](#)

THPlayerController
 adjustRate method, [33](#)
 class extension, [113](#)
 implementation, [115](#)
 initialization, [31](#)
 interface, [30, 113](#)
 play method, [32](#)
 stop method, [33](#)
 volume and panning methods, [34](#)

THPlayerControllerDelegate, [38](#)

THPlayerView, [111](#)

THPreviewView, [177, 220, 234](#)

THRecorderController
 class extension, [48](#)
 formattedCurrentTime method, [51](#)
 init Method, [48](#)
 interface, [47](#)
 levels method, [55](#)
 meter table setup, [54](#)
 playback method, [51](#)
 save method, [50](#)

transport methods, [49](#)
THSampleDataFilter, [271](#)
THSampleDataProvider, [267-268](#)
THSpeechController.h, [20](#)
THSpeechController.m, [21](#)
THTitleItem, [369-371](#)
THTrackMetadataConverter, [89](#)
THTransitionComposition, [348-349](#)
THTransitionCompositionBuilder, [350-351](#)
THTransport.h, [114](#)
THWaveformView Interface, [273](#)
titleForAsset method implementation, [150](#)
Track Contents, [315](#)
Transforming Metadata, [223, 236](#)
Transport Delegate Callbacks, [122](#)
Trimming, [157](#)
validateUserInterfaceItem: method implementation, [158](#)
Video Layers, [379](#)
Video Recording Transport Methods, [203](#)
Visualizing Roll and Yaw, [226](#)
Visualizing the Detected Faces, [224](#)
Writing the Captured Video, [206](#)
Zoom Ramping, [213](#)
Zoom State Observers, [214](#)
loadAudioSamplesFromAsset:completionBlock: class method, [267](#)
loading properties in AVPlayerItem, [116](#)
loadMediaOptions Implementation (listing), [132](#)
loadMediaOptions Set Up (listing), [131](#)
locked exposure mode, [193](#)
Lockwood, Nick, [222, 363](#)
looping audio, [29-34](#)
 configuring audio sessions, [34-36](#)
 handling interruptions, [36-42](#)

responding to route changes, [40-42](#)
lossless codecs, [15](#)
lossy codecs, [15](#)
LPCM (linear pulse-code modulation), [10](#)
luma channel, [13](#)

M

.m4a file extension, [69](#)
.m4b file extension, [69](#)
.m4p file extension, [69](#)
.m4v file extension, [69](#)
machine-readable code detection, [228-241](#)

Mac OS X

audio environment, [25-26](#)
AV Kit framework, [140](#)
capturing media, iOS versus, [169](#)
iTunesLibrary framework, [62-63](#)
media environment, AV Foundation position in, [4-6](#)

main encoding profile

MainViewController Time Polling (listing), [52](#)

makeExportable method, [328, 349](#)
makeFadeInFadeOutAnimation, [373](#)
makePlayable method, [328, 349, 380](#)
managed audio environment (iOS), [26](#)

mdat atom, [66](#)

media. *See also* [audio](#); [video](#)

analog versus digital, [7](#)
audio waveform view, building
 overview, [265-266](#)
 reading audio samples, [266-270](#)
 reducing audio samples, [271-273](#)
 rendering audio samples, [273](#)

capture recording

AVAssetWriter graph, [284-287](#)
capture output delegate, [280-281](#)
overview, [276-277](#)
sample buffer processing, [287-289](#)
session outputs, configuring, [278-280](#)
stopWriting method, [289-290](#)
THCameraController interface, [278](#)
THMovieWriter example, [290-292](#)
THMovieWriter interface, [281-282](#)
THMovieWriter life-cycle methods, [282-285](#)
capturing, [7](#)
 AVCaptureConnection, [172](#)
 AVCaptureDevice, [170-171](#)
 AVCaptureDeviceInput, [171](#)
 AVCaptureOutput, [171](#)
 AVCaptureSession, [170](#)
 AVCaptureVideoPreviewLayer, [172-173](#)
 classes, [170](#)
 CMSampleBuffer, [249-257](#)
 CubeKamera project, [252-257](#)
 face detection, [216-228](#)
 high frame rate video, [241-247](#)
 iOS versus Mac OS X, [169](#)
 Kamera project, [175-208](#)
 machine-readable code detection, [228-241](#)
 processing video, [247-248](#)
 sample code, [174](#)
 video zooming, [209-216](#)
composing, [297-300](#)
 building compositions, [303-307](#)
 exporting compositions, [316-321](#)
 FifteenSeconds project, [307-316](#)
container formats, [18-19](#)

editing, [7](#). *See also* [composing media](#)

inspecting, [6](#)

metadata, [65](#)

MP3 format, [69-70](#)

MPEG-4 format, [68-69](#)

QuickTime format, [66-68](#)

processing, [7](#)

reading, [259](#)

AVAssetReader, [260-261](#)

basic example, [262-265](#)

resetting, [77](#)

writing

AVAssetWriter class, [260-262](#)

basic example, [262-265](#)

interleaving, [261](#)

overview, [259](#)

media environment, AV Foundation position in, [4-6](#)

MediaPlayer framework, [62](#), [135](#), [137](#)

menus, customizing, [153](#)

metadata, [65](#)

attachments, processing video, [251-252](#)

finding, [72](#)

formats

MP3, [69-70](#)

MPEG-4, [68-69](#)

QuickTime, [66-68](#)

headers in video files, [202](#)

KitTime Player project, [150-151](#)

MetaManager app, [76](#)

artwork conversion, [86-87](#)

comment conversion, [87-88](#)

data conversion, [84-86](#)

disc data conversion, [91-93](#)

genre data conversion, [93-96](#)
saving metadata, [98-100](#)
THMediaItem interface, [77-81](#)
THMetadata class, [81-84](#), [96-98](#)
track data conversion, [88-91](#)
retrieving, [70-72](#)
 key/value pairs, [73-75](#)
timed metadata, [151-157](#)
transforming, [223](#), [236](#)

metadataItems method (listing), 96

MetaManager app project, 76

- artwork conversion, [86-87](#)
- comment conversion, [87-88](#)
- data conversion, [84-86](#)
- disc data conversion, [91-93](#)
- genre data conversion, [93-96](#)
- saving metadata, [98-100](#)
- THMediaItem interface, [77-81](#)
- THMetadata class, [81-84](#), [96-98](#)
- track data conversion, [88-91](#)

metering audio, 29, 52-57

microphones, dynamic, 8-9

Minimal control style, 145

mixing audio

- 15 Seconds app, [327-333](#)
 - buildAudioMixWithTrack: method, [331-332](#)
 - Settings menu—audio controls, [333](#)
 - THAudioMixComposition, [327-328](#)
 - THCompositionBuilder, [328-331](#)
 - THVolumeAutomation, [331](#)
- automated volume changes, [324-327](#)
- AVAudioMix, illustration, [324](#)
- AVAudioMixInputParameters, [324](#)

AVMutableAudioMixInputParameters, [325-326](#)
overview, [323-324](#)

modes for audio session categories, 27

Modifying observeValueForKeyPath: (listing), 154

monitorExportProgress method (listing), 319

Monitoring the Export Progress (listing), 319

moov atom, 66

Movie Modernization Preparation (listing), 162

movies. See [video](#)

MP3

- data, [18](#)
- metadata format, [69-70](#)

.mp4 file extension, 69

MPEG-4

- container format, [19](#)
- metadata format, [68-69](#)

MPEG compression, 16

MPMediaPropertyPredicate, 62

MPMoviePlayerController, 137

MPMoviePlayerViewController, 137

MPVolumeView, 135

multiple properties of assets, asynchronous loading, 65

Multi-Route audio session category, 27

mutable AVMetadataItems, 86

N

named voices, 22

Netflix, 3

nextChapter method (listing), 156

nondestructive, defined, 297

None control style, 146

nonlinear, defined, 297

notifications

audio sessions, [37](#)
of route changes, [40](#)
NSAttributedString, [371](#)
NSDictionary, [270](#), [286](#)
NSMenu, [153](#)
NSPredicate, [63](#)
NSSpeechSynthesizer, [19](#)
NSTimeInterval, [51](#), [300](#)
NSTimer, [52](#)
Nyquist rate, [11](#)

O

Observer pattern, [108](#)
observeValueForKeyPath method (listing), [154](#)
observing
item end, [121-122](#)
status changes, [108](#), [117-118](#)
time
 boundary time observation, [119-120](#)
 periodic time observation, [118-119](#)
Observing the status Property (listing), [117](#)
OpenGLTextureCache (listing), [255](#)
OpenGL ES Textures (listing), [256](#)
OpenGL ES video processing, [252-257](#)
options for audio session categories, [27](#)
output. See [input/output](#)
overlapping regions, defining, [340-341](#)

P

panning, controlling in audio player, [29](#)
Panning Method (listing), [34](#)
pass-through time ranges, calculating, [341-344](#)
pause method, [29](#)
PDF-417 codes, [231](#)

peakPowerForChannel method, [53](#)
periodic time observation, [118-119](#)
Periodic Time Observations (listing), [119](#)
P-frames, [17](#)
photos, capturing, [197-199](#)
Play and Record audio session category, [27](#)
playback
 audio, [6](#), [28-30](#)
 with AVSynchronizedLayer, [364-366](#)
 Core Animation in, [380-381](#)
 video, [6](#)
 AirPlay functionality, [133-135](#)
 AV Kit. See [AV Kit](#)
 AVPlayer, [104](#)
 AVPlayerItem, [107](#)
 AVPlayerLayer, [105-106](#)
 boundary time observation, [119-120](#)
 classes, [104](#)
 creating video controller, [113-116](#)
 creating video view, [111-113](#)
 creating visual scrubber, [124-129](#)
 item end observation, [121-122](#)
 keyboard shortcuts, [147](#)
 observing status changes, [117-118](#)
 periodic time observation, [118-119](#)
 sample code, [107-109](#)
 showing subtitles, [129-133](#)
 transport delegate callbacks, [122-124](#)
 Video Player project, [110-118](#)
Playback audio session category, [27](#)
playback method, [51](#)
playback rate, controlling in audio player, [29](#)
playback stack setup, [147-151](#)

play method, [29](#)
play Method Implementation (listing), [32](#)
pointForCaptureDevicePointOfInterest method, [179](#)
predicted frames, [17](#)
prepareToPlay method, [29](#)
prepareToRecord method, [43](#)
prepareWithCompletionHandler: Implementation (listing), [79](#)
preparing composition
 building video layers, [379](#)
 Core Animation
 in export, [381-383](#)
 in playback, [380-381](#)
 THOverlayComposition interface, [378](#)
previews, capturing media, [172-173](#), [176-179](#)
previousChapter method (listing), [156](#)
privacy requirements, capturing media, [185-186](#)
Private THQualityOfService Class (listing), [243](#)
processing
 media, [7](#)
 sample buffers, [287-289](#)
 video, [247-248](#)
 CMSSampleBuffer, [249-257](#)
 CubeKamera project, [252-257](#)
Processing the Sample Buffers (listing), [287-288](#)
processSampleBuffer: method, [287-288](#)
projects
 audio looper, [30-34](#)
 configuring audio sessions, [34-36](#)
 handling interruptions, [36-42](#)
 responding to route changes, [40-42](#)
 CodeKamera, [231-241](#)
 CubeKamera, [252-257](#)
 FaceKamera, [216-228](#)

FifteenSeconds, [307-310](#)
building composition, [311-316](#)
exporting composition, [316-321](#)
view controllers, [308-310](#)

Hello AV Foundation, [19-23](#)

Kamera, [175-208](#)
adjusting flash and torch modes, [195-197](#)
adjusting focus and exposure, [190-195](#)
capturing still images, [197-199](#)
capturing videos, [202-208](#)
configuring cameras, [189-190](#)
configuring capture session, [181-184](#)
creating capture controller, [179-181](#)
creating preview view, [176-179](#)
privacy requirements, [185-186](#)
starting/stopping capture session, [184-185](#)
switching cameras, [186-189](#)
writing to Assets Library, [199-202](#)

KitTime Player, [140-144](#)
chapters, [151-157](#)
converting legacy codecs, [161-165](#)
enabling trimming, [157-159](#)
exporting trimmed video, [159-161](#)
metadata, [150-151](#)
playback stack setup, [147-151](#)

MetaManager app, [76](#)
artwork conversion, [86-87](#)
comment conversion, [87-88](#)
data conversion, [84-86](#)
disc data conversion, [91-93](#)
genre data conversion, [93-96](#)
saving metadata, [98-100](#)

THMediaItem interface, [77-81](#)

THMetadata class, [81-84](#), [96-98](#)
track data conversion, [88-91](#)
SlowKamera, [242-247](#)
starter versus final versions, [19](#)
Video Player, [110-118](#)
 creating video controller, [113-116](#)
 creating video view, [111-113](#)
 observing status changes, [117-118](#)
Voice Memo, [45-52](#)
 configuring audio sessions, [46-52](#)
 enabling audio metering, [52-57](#)
 implementation, [47-52](#)

properties

 of assets, asynchronous loading, [63-65](#)
 loading in AVPlayerItem, [116](#)

ProRes, [17-18](#)

protocol for THPlayerControllerDelegate (listing), [38](#)
pull model, [264](#)

push transitions, [357-359](#)

Q–R

QR codes, [230](#)
QTKit, [162-165](#)
QTMovieModernizer, [162-165](#)
Quartz, [57](#)
queue management of assets, [104](#)
QuickTime, [3](#), [19](#). *See also* [video](#)

 metadata format, [66-68](#)
 QTMovieModernizer, [162-165](#)

readAudioSamplesFromAsset: method, [268-270](#)

reading media, [259](#)

 audio samples, [266-270](#)
 readAudioSamplesFromAsset:, [268-270](#)

THSampleDataProvider, [267](#)
audio waveform view, building
 overview, [265-266](#)
 reading audio samples, [266-270](#)
 reducing audio samples, [271-273](#)
 rendering audio samples, [273](#)

AVAssetReader class
 explained, [260-261](#)
 illustration, [260](#)
basic example, [262-265](#)

capture recording
 AVAssetWriter graph, [284-287](#)
 capture output delegate, [280-281](#)
 overview, [276-277](#)
 sample buffer processing, [287-289](#)
 session outputs, configuring, [278-280](#)
 stopWriting method, [289-290](#)
 THCameraController interface, [278](#)
 THMovieWriter example, [290-292](#)
 THMovieWriter interface, [281-282](#)
 THMovieWriter life-cycle methods, [282-285](#)

Reading the Asset's Audio Samples (listing), [268-270](#)

readyForMoreMediaData property (AVAssetWriterInput), [261-262](#)

Record audio session category, [27](#)

recording audio, [6, 42-45](#)

reducing audio samples, [271-273](#)

reference frames, [17](#)

Register for Route Change Notifications (listing), [40](#)

Registering for Interruption Notifications (listing), [37](#)

rendering

 audio samples, [273](#)
 drawRect: method, [275-276](#)
 setAsset: method, [274](#)

THWaveformView, [273](#)
contexts, [252](#)
renderScale property, [347](#)
renderSize property
 AVMutableVideoComposition, [346](#)
 AVVideoComposition, [347](#)
requestMediaDataWhenReadyOnQueue:usingBlock: method, [262](#)
resetFocusAndExposureModes method (listing), [194](#)
Resetting Focus and Exposure (listing), [194](#)
resetting media, [77](#)
retrieving metadata, [70-72](#)
 key/value pairs, [73-75](#), [81-84](#)
 MetaManager app project, [77-81](#)
roll angle, [216](#), [226](#)
route changes, responding to, [40-42](#)
route picker for AirPlay, [134-135](#)
Running the Modernization (listing), [163](#)

S

sample buffers, processing, [287-289](#)
sampling, [7-8](#)
 audio sampling, [8-13](#)
 spatial sampling, [8](#)
 temporal sampling, [8](#)
sampling rate, [10-13](#), [44](#)
save method, [50](#)
saveWithCompletionHandler: Implementation (listing), [99](#)
saving
 compositions, [299](#)
 metadata, [98-100](#)
scanning barcodes, [228-241](#)
screen versus capture device coordinates, [178-179](#)
scrubbers, creating visual scrubber, [124-129](#)

Scrubbing Methods (listing), [123](#)
serial dispatch queues, [119](#)
serial queues, [254](#)
session outputs, configuring, [278-280](#)
setAsset: method, [274](#)
Settings menu (audio controls), [333](#)
Setting Up the AVAssetWriter Graph (listing), [284-286](#)
Setting up the Playback Stack (listing), [148](#)
setupSession: Method (listing), [181](#)
setupView Method implementation (listing), [222](#)
setVolume:atTime: method, [325-326](#)
setVolumeRampFromStartVolume:toEndVolume:timeRange: method, [325-326](#)
Skype, [3](#)
SlowKamera project, [242-247](#)
slow motion with high frame rate video capture, [241-247](#)
smooth focus mode, [205](#)
Solo Ambient audio session category, [26, 34](#)
sound. See [audio](#)
spatial sampling, [8](#)
speech synthesizer project (Hello AV Foundation), [19-23](#)
staggering video layout, [338-340](#)
starter versions of projects, [19](#)
startExporting method implementation (listing), [160](#)
starting
 capture sessions, [184-185](#)
 video recording, [203](#)
Starting and Stopping the Capture Session (listing), [184](#)
startReading method, [263](#)
startSessionAtSourceTime: method, [265, 288](#)
startSession method, [184](#)
startWriting method, [263](#)
status changes, observing, [108, 117-118](#)

status Property (listing), [117](#)
still images, capturing, [197-199](#)
stop method, [29](#)
stop Method Implementation (listing), [33](#)
stopping
 capture sessions, [184-185](#)
 video recording, [203](#)
Stop Playback on Headphone Unplug (listing), [42](#)
stopSession method, [184](#)
stopWriting method, [289-290](#)
storage requirements
 audio, [12](#)
 video, [13](#)
subtitles
 enabling, [133](#)
 showing, [129-133](#)
subtracting time, [301](#)
switching cameras, [186-189](#)
Switching Cameras (listing), [188](#)

T

TangoMe, [3](#)
Tap-to-Expose Methods (listing), [192](#)
Tap-to-Focus Implementation (listing), [190](#)
temporal sampling, [8](#)
TH720pVideoRect, [371](#)
THAppDelegate Audio Session Setup (listing), [35, 46](#)
THArtworkMetadataConverter Implementation (listing), [86](#)
THAudioMixComposition
 implementation, [327-328](#)
 interface, [327](#)
THAudioMixCompositionBuilder
 implementation, [329-331](#)

interface, [329](#)

THBasicComposition, [311](#)

THBasicCompositionBuilder, [313](#)

THCameraController

implementation, [212](#), [217](#)

interface, [180](#), [211](#), [217](#), [231](#), [252](#), [278](#)

THCommentMetadataConverter Implementation (listing), [87](#)

THComposition Protocol (listing), [311](#)

THCompositionBuilder, [328-331](#), [349-351](#)

THCompositionBuilder Protocol (listing), [313](#)

THCompositionExporter Interface (listing), [317](#)

THDefaultMetadataConverter Implementation (listing), [85](#)

THDiscMetadataConverter Implementation (listing), [91](#)

THDocument Implementation (listing), [143](#)

THFilterSelectionChangedNotification, [284](#)

THGenreMetadataConverter Implementation (listing), [94](#)

THMainViewController Metering Methods (listing), [56](#)

THMediaItem, [77-81](#), [368](#)

implementation, [78](#)

interface, [77](#)

saveWithCompletionHandler: implementation, [99](#)

THMetadata

implementation, [82](#)

MetaManager app project, [81-84](#), [96-98](#)

THMetadataConverter Interface Template (listing), [85](#)

THMetadataConverter Protocol (listing), [85](#)

THMetadataItem Interface (listing), [81](#)

THMeterTable Implementation (listing), [53](#)

THMovieWriter

example, [290-292](#)

interface, [281-282](#)

life-cycle methods, [282-285](#)

THOverlayComposition, [380](#)

interface, [378](#)

THPlayerController

adjustRate method implementation, [33](#)
class extension, [113](#)
implementation, [115](#)
initialization, [31](#)
interface, [30](#), [113](#)
play method implementation, [32](#)
stop method implementation, [33](#)
volume and panning methods, [34](#)

THPlayerView, [111](#)

THPreviewView, [177](#), [220](#), [234](#)

THQualityOfService Class (listing), [243](#)

THRecorderController

class extension, [48](#)
formattedCurrentTime method, [51](#)
init method, [48](#)
interface, [47](#)
levels method, [55](#)
meter table setup, [54](#)
playback method, [51](#)
save method, [50](#)
transport methods, [49](#)

THSampleDataFilter, [271-273](#), [276](#)

implementation, [271-272](#)
interface, [271](#)

THSampleDataProvider, [267-268](#)

implementation, [267-268](#)
interface, [267](#)

THSpeechController.h (listing), [20](#)

THSpeechController.m (listing), [21](#)

THTimelineItem, [368](#), [373](#)

THTitleItem, [368-372](#)

building layers, [369-371](#)
interface, [369](#)

THTrackMetadataConverter Implementation (listing), 89

THTransitionComposition

implementation, [348-349](#)
interface, [348](#)

THTransitionCompositionBuilder

implementation, [350-351](#)
interface, [350](#)

THTransport.h (listing), 114

THVolumeAutomation, 331

THWaveformView, 273

time

CMTTime, [109-110](#), [300-301](#)
CMTTimeRange, [302-303](#)
as floating-point value, [109](#)
observing
 boundary time observation, [119-120](#)
 periodic time observation, [118-119](#)

time display in audio recorder, 51

time ranges

pass-through time ranges, calculating, [341-344](#)
transition time ranges, calculating, [341-344](#)

timed metadata, 151-157

timing information, processing video, 251

timing model for Core Animation, 363-364

titleForAsset method (listing), 150

title image animation, 373-375

titles, animated, 367-378

 data model, [368](#)
 fade in/fade out animation, [372-373](#)
 image animation methods, [375-378](#)
 THTitleItem, [369-372](#)

title image animation, [373-375](#)
torch mode, adjusting, [195-197](#)
toStartTransform, [358](#)
track contents, building, [315](#)
track data conversion, [88-91](#)
tracks of assets, [60](#)
transducers, [8](#)
transformation matrix, [222](#)
Transforming Metadata (listing), [223](#), [236](#)
transitionDuration, [340](#)
transition effects, dissolve transitions, [357](#)
transition instructions, extracting, [355-356](#)
transitionInstructionsInVideoComposition: method, [355-356](#)
transition time ranges, calculating, [341-344](#)
transitions. *See* [video transitions](#)
transport delegate callbacks, [122-124](#)
Transport Delegate Callbacks (listing), [122](#)
transport methods, [49](#)
trimming
 enabling, [157-159](#)
 exporting trimmed video, [159-161](#)

U

UIKit framework, [4](#)
UISlider, [377](#)
UIView, [111](#), [113](#)
UIViewController, [113](#), [138](#)
UIWebView framework, [4](#)
unretained references, [268](#)
UPC-E barcodes, [229](#)
URLs, creating assets, [60](#)
user data (QuickTime), [67](#)
Using Core Animation in Export (listing), [381-382](#)

Using Core Animation in Playback (listing), [380](#)

Using the THMovieWriter (listing), [290-292](#)

utterances, [19](#)

V

validateUserInterfaceItem: method implementation (listing), [158](#)
video. See also [media](#)

capturing in Kamera project, [202-208](#). *See also* [capturing media](#)
chroma subsampling, [13-15](#)

Core Video, [5](#)

frames, [12](#)

high frame rate video, [241-247](#)

playback, [6](#)

AirPlay functionality, [133-135](#)

AV Kit. *See* [AV Kit](#)

AVPlayer, [104](#)

AVPlayerItem, [107](#)

AVPlayerLayer, [105-106](#)

boundary time observation, [119-120](#)

classes, [104](#)

creating video controller, [113-116](#)

creating video view, [111-113](#)

creating visual scrubber, [124-129](#)

item end observation, [121-122](#)

keyboard shortcuts, [147](#)

observing status changes, [117-118](#)

periodic time observation, [118-119](#)

sample code, [107-109](#)

showing subtitles, [129-133](#)

transport delegate callbacks, [122-124](#)

Video Player project, [110-118](#)

processing, [247-248](#)

CMSampleBuffer, [249-257](#)

CubeKamera project, [252-257](#)
storage requirements, [13](#)
timescales, [301](#)
zooming, [209-216](#)

video codecs, [15-18](#)

video controllers, creating, [113-116](#)

video gravities, [172](#)

video gravity values, [105](#)

video layers, building, [379](#)

video layout, staggering, [338-340](#)

Video Player project, [110-118](#)
creating video controller, [113-116](#)
creating video view, [111-113](#)
observing status changes, [117-118](#)

Video Recording Transport Methods (listing), [203](#)

video stabilization, [205](#)

video transitions
15 Seconds app
buildCompositionTracks method, [351-353](#)
buildVideoComposition: method, [353-355](#)
THCompositionBuilder, [349-351](#)
THTransitionComposition, [348-349](#)
transitionInstructionsInVideoComposition: method, [355-356](#)

AVVideoComposition, [336](#)
AVVideoCompositionInstruction, [336](#)
AVVideoCompositionLayerInstruction, [337](#)
conceptual steps, [337](#)
building and configuring AVVideoComposition, [346-347](#)
building composition and layer instructions, [344-346](#)
calculating pass-through and transition time ranges, [341-344](#)
defining overlapping regions, [340-341](#)
staggering video layout, [338-340](#)

overview, [335](#)

push transitions, [357-359](#)
videoCompositionWithPropertiesOfAsset: method, [347-348](#)
wipe transitions, [359-360](#)

videoCompositionWithPropertiesOfAsset: method, [347-348](#)

videoGravity property, [105](#)

videoScaleAndCropFactor property, [209](#)

videoSupportedFrameRateRanges property, [242](#)

videoZoomFactor property, [209-216](#)

view controllers, [308-310](#)

Visualizing Roll and Yaw (listing), [226](#)

Visualizing the Detected Faces (listing), [224](#)

visual scrubber, creating, [124-129](#)

Voice Memo project, [45-52](#)

- configuring audio sessions, [46-52](#)
- enabling audio metering, [52-57](#)
- implementation, [47-52](#)

volume

- automated volume changes, [324-327](#)
- controlling in audio player, [29](#)

Volume Method (listing), [34](#)

W

WebView framework, [4](#)

wipe transitions, [359-360](#)

writing media, [259](#)

- audio waveform view, building
 - overview, [265-266](#)
 - reading audio samples, [266-270](#)
 - reducing audio samples, [271-273](#)
 - rendering audio samples, [273](#)

AVAssetWriter class

- explained, [261-262](#)
- illustration, [260](#)

basic example, [262-265](#)

capture recording

AVAssetWriter graph, [284-287](#)

capture output delegate, [280-281](#)

overview, [276-277](#)

sample buffer processing, [287-289](#)

session outputs, configuring, [278-280](#)

stopWriting method, [289-290](#)

THCameraController interface, [278](#)

THMovieWriter, [290-292](#)

THMovieWriter interface, [281-282](#)

THMovieWriter life-cycle methods, [282-285](#)

interleaving, [261](#)

writing sessions, finishing, [289-290](#)

Writing the Captured Video (listing), [206](#)

writing to Assets Library framework, [199-202](#)

X–Y–Z

yaw angle, [216, 226](#)

Y'CbCr color model, [13](#)

YouTube, [3](#)

YUV color model, [13](#)

zooming video, [209-216](#)



REGISTER



THIS PRODUCT

informit.com/register

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account.

You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE



THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

InformIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the leading brands, authors, and contributors from the tech community.

▼Addison-Wesley **Cisco Press** **IBM
Press** Microsoft Press

PEARSON IT CERTIFICATION PRENTICE HALL QUE SAMS VMWARE PRESS

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials. Looking for expert opinions, advice, and tips? **InformIT has a solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of monthly newsletters. Visit informit.com/newsletters.
- FREE Podcasts from experts at informit.com/podcasts.
- Read the latest author articles and sample chapters at informit.com/articles.
- Access thousands of books and videos in the Safari Books Online digital library. safari.informit.com.
- Get Advice and tips from expert blogs at informit.com/blogs.

Visit informit.com to find out all the ways you can access the hottest technology content.

Are you part of the **IT** crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube and more! Visit informit.com/socialconnect.



ALWAYS LEARNING

PEARSON

Code Snippets

```
AVSpeechSynthesizer *synthesizer = [[AVSpeechSynthesizer alloc] init];
AVSpeechUtterance *utterance =
    [[AVSpeechUtterance alloc] initWithString:@"Hello World!"];
[synthesizer speakUtterance:utterance];
```

```
#import <AVFoundation/AVFoundation.h>

@interface THSpeechController : NSObject

@property (strong, nonatomic, readonly) AVSpeechSynthesizer *synthesizer;
+ (instancetype)speechController;
- (void)beginConversation;

@end
```

```
#import "THSpeechController.h"
#import <AVFoundation/AVFoundation.h>

@interface THSpeechController ()
@property (strong, nonatomic) AVSpeechSynthesizer *synthesizer; // 1
@property (strong, nonatomic) NSArray *voices;
@property (strong, nonatomic) NSArray *speechStrings;
@end

@implementation THSpeechController

+ (instancetype)speechController {
    return [[self alloc] init];
}

- (id)init {
    self = [super init];
    if (self) {
        _synthesizer = [[AVSpeechSynthesizer alloc] init]; // 2
        _voices = @[[AVSpeechSynthesisVoice voiceWithLanguage:@"en-US"],
                    [AVSpeechSynthesisVoice voiceWithLanguage:@"en-GB"]];
        _speechStrings = [self buildSpeechStrings];
    }
    return self;
}

- (NSArray *)buildSpeechStrings { // 4
    return @[@"Hello AV Foundation. How are you?",
             @"I'm well! Thanks for asking.",
             @"Are you excited about the book?",
             @"Very! I have always felt so misunderstood",
             @"What's your favorite feature?",
             @"Oh, they're all my babies. I couldn't possibly choose.",
             @"It was great to speak with you!",
             @"The pleasure was all mine! Have fun!"];
}

- (void)beginConversation {

}

@end
```

```
- (void)beginConversation {
    for (NSUInteger i = 0; i < self.speechStrings.count; i++) {
        AVSpeechUtterance *utterance =
            [[AVSpeechUtterance alloc] initWithString:self.speechStrings[i]]; // 1
        utterance.voice = self.voices[i % 2]; // 2
        utterance.rate = 0.4f; // 3
        utterance.pitchMultiplier = 0.8f; // 4
        utterance.postUtteranceDelay = 0.1f; // 5
        [self.synthesizer speakUtterance:utterance];
    }
}
```

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    AVAudioSession *session = [AVAudioSession sharedInstance];
    NSError *error;
    if (![session setCategory:AVAudioSessionCategoryPlayback error:&error]) {
        NSLog(@"%@", [error localizedDescription]);
    }
    if (![session setActive:YES error:&error]) {
        NSLog(@"%@", [error localizedDescription]);
    }
    return YES;
}
```

```
NSURL *fileURL =
[[NSBundle mainBundle] URLForResource:@"rock" withExtension:@"mp3"] ;

// Must maintain a strong reference to player
self.player =
[[AVAudioPlayer alloc] initWithContentsOfURL:fileURL error:nil];

if (self.player) {
    [self.player prepareToPlay];
}
```

```
@interface THPlayerController : NSObject

@property (nonatomic, readonly, getter = isPlaying) BOOL playing;

// Global methods
- (void)play;
- (void)stop;
- (void)adjustRate:(float)rate;

// Player-specific methods
- (void)adjustPan:(float)pan forPlayerAtIndex:(NSUInteger)index;
- (void)adjustVolume:(float)volume forPlayerAtIndex:(NSUInteger)index;

@end
```

```
#import "THPlayerController.h"
#import <AVFoundation/AVFoundation.h>

@interface THPlayerController ()
@property (nonatomic) BOOL playing;
@property (strong, nonatomic) NSArray *players;
@end

@implementation THPlayerController

- (instancetype)init {
    self = [super init];
    if (self) {
        AVAudioPlayer *guitarPlayer = [self playerForFile:@"guitar"];
        AVAudioPlayer *bassPlayer = [self playerForFile:@"bass"];
        AVAudioPlayer *drumsPlayer = [self playerForFile:@"drums"];
        _players = @*[guitarPlayer, bassPlayer, drumsPlayer];
    }
    return self;
}

- (AVAudioPlayer *)playerForFile:(NSString *)name {
    NSURL *fileURL = [[NSBundle mainBundle] URLForResource:name
                      withExtension:@"caf"];
    NSError *error;
    AVAudioPlayer *player = [[AVAudioPlayer alloc] initWithContentsOfURL:fileURL
                                                               error:&error];
    if (player) {
        player.numberOfLoops = -1; // loop indefinitely
        player.enableRate = YES;
        [player prepareToPlay];
    } else {
        NSLog(@"Error creating player: %@", [error localizedDescription]);
    }

    return player;
}

```

```
- (void)play {
    if (!self.playing) {
        NSTimeInterval delayTime = [self.players[0] deviceCurrentTime] + 0.01;
        for (AVAudioPlayer *player in self.players) {
            [player playAtTime:delayTime];
        }
        self.playing = YES;
    }
}
```

```
- (void)stop {
    if (self.isPlaying) {
        for (AVAudioPlayer *player in self.players) {
            [player stop];
            player.currentTime = 0.0f;
        }
        self.isPlaying = NO;
    }
}
```

```
- (void)adjustRate:(float)rate {
    for (AVAudioPlayer *player in self.players) {
        player.rate = rate;
    }
}
```

```
- (void)adjustPan:(float)pan forPlayerAtIndex:(NSUInteger)index {
    if ([self isValidIndex:index]) {
        AVAudioPlayer *player = self.players[index];
        player.pan = pan;
    }
}

- (void)adjustVolume:(float)volume forPlayerAtIndex:(NSUInteger)index {
    if ([self isValidIndex:index]) {
        AVAudioPlayer *player = self.players[index];
        player.volume = volume;
    }
}

- (BOOL)isValidIndex:(NSUInteger)index {
    return index == 0 || index < self.players.count;
}
```

```
#import "THAppDelegate.h"
#import <AVFoundation/AVFoundation.h>

@implementation THAppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    AVAudioSession *session = [AVAudioSession sharedInstance];

    NSError *error;
    if (![session setCategory:AVAudioSessionCategoryPlayback error:&error]) {
        NSLog(@"%@", [error localizedDescription]);
    }
    if (![session setActive:YES error:&error]) {
        NSLog(@"%@", [error localizedDescription]);
    }

    return YES;
}

@end
```

```
<key>UIBackgroundModes</key>
<array>
    <string>audio</string>
</array>
```

```
- (instancetype)init {
    self = [super init];
    if (self) {
        AVAudioPlayer *guitarPlayer = [self playerForFile:@"guitar"];
        AVAudioPlayer *bassPlayer = [self playerForFile:@"bass"];
        AVAudioPlayer *drumsPlayer = [self playerForFile:@"drums"];

        _players = @[_guitarPlayer, bassPlayer, drumsPlayer];

        NSNotificationCenter *nsnc = [NSNotificationCenter defaultCenter];
        [nsnc addObserver:self
                    selector:@selector(handleInterruption:)
                    name:AVAudioSessionInterruptionNotification
                    object:[AVAudioSession sharedInstance]];
    }
    return self;
}

- (void)dealloc {
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}

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

```
- (void)handleInterruption: (NSNotification *)notification {  
  
    NSDictionary *info = notification.userInfo;  
  
    AVAudioSessionInterruptionType type =  
        [info [AVAudioSessionInterruptionTypeKey] unsignedIntegerValue];  
  
    if (type == AVAudioSessionInterruptionTypeBegan) {  
        // Handle AVAudioSessionInterruptionTypeBegan  
    } else {  
        // Handle AVAudioSessionInterruptionTypeEnded  
    }  
}
```

```
@protocol THPlayerControllerDelegate <NSObject>
- (void)playbackStopped;
- (void)playbackBegan;
@end
```

```
- (void)handleInterruption:(NSNotification *)notification {
    NSDictionary *info = notification.userInfo;
    AVAudioSessionInterruptionType type =
        [info[AVAudioSessionInterruptionTypeKey] unsignedIntegerValue];
    if (type == AVAudioSessionInterruptionTypeBegan) {
        [self stop];
        if (self.delegate) {
            [self.delegate playbackStopped];
        }
    } else {
        // Handle interruption ended
    }
}
```

```
- (void)handleInterruption:(NSNotification *)notification {
    NSDictionary *info = notification.userInfo;
    AVAudioSessionInterruptionType type =
        [info[AVAudioSessionInterruptionTypeKey] unsignedIntegerValue];
    if (type == AVAudioSessionInterruptionTypeBegan) {
        [self stop];
        if (self.delegate) {
            [self.delegate playbackStopped];
        }
    } else {
        AVAudioSessionInterruptionOptions options =
            [info[AVAudioSessionInterruptionOptionKey] unsignedIntegerValue];
        if (options == AVAudioSessionInterruptionOptionShouldResume) {
            [self play];
            if (self.delegate) {
                [self.delegate playbackBegan];
            }
        }
    }
}
```

```
- (instancetype)init {
    self = [super init];
    if (self) {
        AVAudioPlayer *guitarPlayer = [self playerForFile:@"guitar"];
        AVAudioPlayer *bassPlayer = [self playerForFile:@"bass"];
        AVAudioPlayer *drumsPlayer = [self playerForFile:@"drums"];

        _players = @[guitarPlayer, bassPlayer, drumsPlayer];

        NSNotificationCenter *nsnc = [NSNotificationCenter defaultCenter];
        [nsnc addObserver:self
                    selector:@selector(handleInterruption:)
                      name:AVAudioSessionInterruptionNotification
                     object:[AVAudioSession sharedInstance]];

        [nsnc addObserver:self
                    selector:@selector(handleRouteChange:)
                      name:AVAudioSessionRouteChangeNotification
                     object:[AVAudioSession sharedInstance]];
    }
    return self;
}

- (void)handleRouteChange:(NSNotification *)notification {  
}
```

```
- (void)handleRouteChange:(NSNotification *)notification {
    NSDictionary *info = notification.userInfo;
    AVAudioSessionRouteChangeReason reason =
        [info[AVAudioSessionRouteChangeReasonKey] unsignedIntValue];
    if (reason == AVAudioSessionRouteChangeReasonOldDeviceUnavailable) {
    }
}
```

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

    NSDictionary *info = notification.userInfo;
    AVAudioSessionRouteChangeReason reason =
        [info[AVAudioSessionRouteChangeReasonKey] unsignedIntValue];

    if (reason == AVAudioSessionRouteChangeReasonOldDeviceUnavailable) {

        AVAudioSessionRouteDescription *previousRoute =
            info[AVAudioSessionRouteChangePreviousRouteKey];

        AVAudioSessionPortDescription *previousOutput = previousRoute.outputs[0];
        NSString *portType = previousOutput.portType;

        if ([portType isEqualToString:AVAudioSessionPortHeadphones]) {
            [self stop];
            [self.delegate playbackStopped];
        }
    }
}
```

```
NSString *directory = // output directory
NSString *filePath = [directory stringByAppendingPathComponent:@"voice.m4a"];
NSURL *url = [NSURL fileURLWithPath:filePath];

NSDictionary *settings = @{@"AVFormatIDKey" : @(kAudioFormatMPEG4AAC),
                           AVSampleRateKey : @22050.0f,
                           AVNumberOfChannelsKey : @1};

NSError *error;
// Must maintain a strong reference to player
self.recorder =
    [[AVAudioRecorder alloc] initWithURL:url settings:settings error:&error];
if (self.recorder) {
    [self.recorder prepareToRecord];
} else {
    // Handle error
}
```

kAudioFormatLinearPCM

kAudioFormatMPEG4AAC

kAudioFormatAppleLossless

kAudioFormatAppleIMA4

kAudioFormatiLBC

kAudioFormatULaw

The operation couldn't be completed. (OSStatus error 1718449215.)

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    AVAudioSession *session = [AVAudioSession sharedInstance];

    NSError *error;
    if (![session setCategory:AVAudioSessionCategoryPlayAndRecord error:&error]) {
        NSLog(@"%@", [error localizedDescription]);
    }

    if (![session setActive:YES error:&error]) {
        NSLog(@"%@", [error localizedDescription]);
    }

    return YES;
}
```

```
typedef void(^THRecordingStopCompletionHandler)(BOOL);
typedef void(^THRecordingSaveCompletionHandler)(BOOL, id);

@class THMemo;

@interface THRecorderController : NSObject

@property (nonatomic, readonly) NSString *formattedCurrentTime;

// Recorder methods
- (BOOL)record;

- (void)pause;

- (void)stopWithCompletionHandler:(THRecordingStopCompletionHandler)handler;

- (void)saveRecordingWithName:(NSString *)name
                      completionHandler:(THRecordingSaveCompletionHandler)handler;
// Player methods
- (BOOL)playbackMemo:(THMemo *)memo;

@end
```

```
@interface THRecorderController () <AVAudioRecorderDelegate>

@property (strong, nonatomic) AVAudioPlayer *player;
@property (strong, nonatomic) AVAudioRecorder *recorder;
@property (strong, nonatomic) THRecordingStopCompletionHandler completionHandler;

@end
```

```
- (instancetype)init {
    self = [super init];
    if (self) {
        NSString *tmpDir = NSTemporaryDirectory();
        NSString *filePath = [tmpDir stringByAppendingPathComponent:@"memo.caf"];
        NSURL *fileURL = [NSURL fileURLWithPath:filePath];

        NSDictionary *settings = @{
            AVFormatIDKey : @(kAudioFormatAppleIMA4),
            AVSampleRateKey : @44100.0f,
            AVNumberOfChannelsKey : @1,
            AVEncoderBitDepthHintKey : @16,
            AVEncoderAudioQualityKey : @(AVAudioQualityMedium)
        };
        NSError *error;
        self.recorder = [[AVAudioRecorder alloc] initWithURL:fileURL
                                                    settings:settings
                                                    error:&error];
        if (self.recorder) {
            self.recorder.delegate = self;
            [self.recorder prepareToRecord];
        } else {
            NSLog(@"Error: %@", [error localizedDescription]);
        }
    }
    return self;
}
```

```
- (BOOL)record {
    return [self.recorder record];
}

- (void)pause {
    [self.recorder pause];
}

- (void)stopWithCompletionHandler:(THRecordingStopCompletionHandler)handler {
    self.completionHandler = handler;
    [self.recorder stop];
}

- (void)audioRecorderDidFinishRecording:(AVAudioRecorder *)recorder
    successfully:(BOOL)success {
    if (self.completionHandler) {
        self.completionHandler(success);
    }
}
```

```
- (void)saveRecordingWithName:(NSString *)name
    completionHandler:(THRecordingSaveCompletionHandler)handler {

    NSTimeInterval timestamp = [NSDate timeIntervalSinceReferenceDate];
    NSString *filename =
        [NSString stringWithFormat:@"%@-%f.caf", name, timestamp];

    NSString *docsDir = [self documentsDirectory];
    NSString *destPath = [docsDir stringByAppendingPathComponent:filename];

    NSURL *srcURL = self.recorder.url;
    NSURL *destURL = [NSURL fileURLWithPath:destPath];

    NSError *error;
    BOOL success = [[NSFileManager defaultManager] copyItemAtURL:srcURL
                                                toURL:destURL
                                              error:&error];
    if (success) {
        handler(YES, [THMemo memoWithTitle:name url:destURL]);
        [self.recorder prepareToRecord];
    } else {
        handler(NO, error);
    }
}

- (NSString *)documentsDirectory {
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                       NSUserDomainMask, YES);
    return [paths objectAtIndex:0];
}
```

```
- (BOOL)playbackMemo:(THMemo *)memo {
    [self.player stop];
    self.player = [[AVAudioPlayer alloc] initWithContentsOfURL:memo.url
                                                error:nil];
    if (self.player) {
        [self.player play];
        return YES;
    }
    return NO;
}
```

```
- (NSString *)formattedCurrentTime {

    NSUInteger time = (NSUInteger)self.recorder.currentTime;
    NSInteger hours = (time / 3600);
    NSInteger minutes = (time / 60) % 60;
    NSInteger seconds = time % 60;

    NSString *format = @"%02i:%02i:%02i";
    return [NSString stringWithFormat:format, hours, minutes, seconds];
}
```

```
- (void)startTimer {
    [self.timer invalidate];
    self.timer = [NSTimer timerWithTimeInterval:0.5
                                         target:self
                                         selector:@selector(updateTimeDisplay)
                                         userInfo:nil
                                         repeats:YES];
    [[NSRunLoop mainRunLoop] addTimer:self.timer forMode:NSRunLoopCommonModes];
}
- (void)updateTimeDisplay {
    self.timeLabel.text = self.controller.formattedCurrentTime;
}
```

```
#import "THMeterTable.h"

#define MIN_DB -60.0f
#define TABLE_SIZE 300

@implementation THMeterTable {
    float _scaleFactor;
    NSMutableArray *_meterTable;
}

- (instancetype)init {
    self = [super init];
    if (self) {
        float dbResolution = MIN_DB / (TABLE_SIZE - 1);

        _meterTable = [NSMutableArray arrayWithCapacity:TABLE_SIZE];
        _scaleFactor = 1.0f / dbResolution;

        float minAmp = dbToAmp(MIN_DB);
        float ampRange = 1.0 - minAmp;
        float invAmpRange = 1.0 / ampRange;
    }
}
```

```
        for (int i = 0; i < TABLE_SIZE; i++) {
            float decibels = i * dbResolution;
            float amp = dbToAmp(decibels);
            float adjAmp = (amp - minAmp) * invAmpRange;
            _meterTable[i] = @(adjAmp);
        }
    }
    return self;
}

float dbToAmp(float dB) {
    return powf(10.0f, 0.05f * dB);
}

- (float)valueForPower:(float)power {
    if (power < MIN_DB) {
        return 0.0f;
    } else if (power >= 0.0f) {
        return 1.0f;
    } else {
        int index = (int) (power * _scaleFactor);
        return [_meterTable[index] floatValue];
    }
}

@end
```

```
@interface THRecorderController () <AVAudioRecorderDelegate>

@property (strong, nonatomic) AVAudioPlayer *player;
@property (strong, nonatomic) AVAudioRecorder *recorder;
@property (strong, nonatomic) THRecordingStopCompletionHandler completionHandler;
@property (strong, nonatomic) THMeterTable *meterTable;

@end

@implementation THRecorderController

- (instancetype)init {
    self = [super init];
    if (self) {
        NSString *tmpDir = NSTemporaryDirectory();
        NSString *filePath = [tmpDir stringByAppendingPathComponent:@"memo.caf"];
        NSURL *fileURL = [NSURL fileURLWithPath:filePath];

        NSDictionary *settings = @{
            AVFormatIDKey : @(kAudioFormatAppleIMA4),
            AVSampleRateKey : @44100.0f,
            AVNumberOfChannelsKey : @1,
            AVEncoderBitDepthHintKey : @16,
            AVEncoderAudioQualityKey : @(AVAudioQualityMedium)
        };

        NSError *error;
        self.recorder = [[AVAudioRecorder alloc] initWithURL:fileURL
                                                    settings:settings
                                                    error:&error];
        if (self.recorder) {
            self.recorder.delegate = self;
            self.recorder.meteringEnabled = YES;
            [self.recorder prepareToRecord];
        } else {
            NSLog(@"Error: %@", [error localizedDescription]);
        }

        _meterTable = [[THMeterTable alloc] init];
    }
}

return self;
}
```

```
- (THLevelPair *)levels {
    [self.recorder updateMeters];
    float avgPower = [self.recorder averagePowerForChannel:0];
    float peakPower = [self.recorder peakPowerForChannel:0];
    float linearLevel = [self.meterTable valueForPower:avgPower];
    float linearPeak = [self.meterTable valueForPower:peakPower];
    return [THLevelPair levelsWithLevel:linearLevel peakLevel:linearPeak];
}
```

```
- (void)startMeterTimer {
    [self.levelTimer invalidate];
    self.levelTimer = [CADisplayLink displayLinkWithTarget:self
                                                selector:@selector(updateMeter)];
    self.levelTimer.frameInterval = 5;
    [self.levelTimer addToRunLoop:[NSRunLoop currentRunLoop]
                           forMode:NSRunLoopCommonModes];
}

- (void)stopMeterTimer {
    [self.levelTimer invalidate];
    self.levelTimer = nil;
    [self.levelMeterView resetLevelMeter];
}

- (void)updateMeter {
    THLevelPair *levels = [self.controller levels];
    self.levelMeterView.level = levels.level;
    self.levelMeterView.peakLevel = levels.peakLevel;
    [self.levelMeterView setNeedsDisplay];
}
```

```
NSURL *assetURL = // url  
AVAsset *asset = [AVAsset assetWithURL:assetURL];
```

```
NSURL *assetURL = // url  
NSDictionary *options = @{@"AVURLAssetPreferPreciseDurationAndTimingKey": @YES};  
AVURLAsset *asset = [[AVURLAsset alloc] initWithURL:assetURL options:options];
```

```
ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];

[library enumerateGroupsWithTypes:ALAssetsGroupSavedPhotos
    usingBlock:^(ALAssetsGroup *group, BOOL *stop) {

    // Filter down to only videos
    [group setAssetsFilter:[ALAssetsFilter allVideos]];

    // Grab the first video returned
    [group enumerateAssetsAtIndexes:[NSSet indexSetWithIndex:0]
        options:0
        usingBlock:^(ALAsset *alAsset,
                    NSUInteger index,
                    BOOL *innerStop) {

        if (alAsset) {
            id representation = [alAsset defaultRepresentation];
            NSURL *url = [representation url];
            AVAsset *asset = [AVAsset assetWithURL:url];
            // Asset created. Perform some AV Foundation magic.
        }
    }];
}

} failureBlock:^(NSError *error) {
    NSLog(@"%@", [error localizedDescription]);
}];
```

```
MPMediaPropertyPredicate *artistPredicate =
[MPMediaPropertyPredicate predicateWithValue:@"Foo Fighters"
                           forProperty:MPMediaItemPropertyArtist];

MPMediaPropertyPredicate *albumPredicate =
[MPMediaPropertyPredicate predicateWithValue:@"In Your Honor"
                           forProperty:MPMediaItemPropertyAlbumTitle];

MPMediaPropertyPredicate *songPredicate =
[MPMediaPropertyPredicate predicateWithValue:@"Best of You"
                           forProperty:MPMediaItemPropertyTitle];

MPMediaQuery *query = [[MPMediaQuery alloc] init];

[query addFilterPredicate:artistPredicate];
[query addFilterPredicate:albumPredicate];
[query addFilterPredicate:songPredicate];

NSArray *results = [query items];
if (results.count > 0) {
    MPMediaItem *item = results[0];
    NSURL *assetURL = [item valueForProperty:MPMediaItemPropertyAssetURL];
    AVAsset *asset = [AVAsset assetWithURL:assetURL];
    // Asset created. Perform some AV Foundation magic.
}
```

```
ITLibrary *library = [ITLibrary libraryWithAPIVersion:@"1.0" error:nil];

NSArray *items = self.library.allMediaItems;
NSString *query = @"artist.name == 'Robert Johnson' AND "
                  "album.title == 'King of the Delta Blues Singers' AND "
                  "title == 'Cross Road Blues'";
NSPredicate *predicate =
    [NSPredicate predicateWithFormat:query];

NSArray *songs = [items filteredArrayUsingPredicate:predicate];
if (songs.count > 0) {
    ITLibMediaItem *item = songs[0];
    AVAsset *asset = [AVAsset assetWithURL:item.location];
    // Asset created. Perform some AV Foundation magic.
}
```

- (AVKeyValueStatus)statusOfValueForKey:(NSString *)key
error:(NSError **)outError
- (void)loadValuesAsynchronouslyForKeys:(NSArray *)keys
completionHandler:(void (^)(void))handler

```
// Create URL for 'sunset.mov' in the application bundle
NSURL *assetURL =
    [[NSBundle mainBundle] URLForResource:@"sunset" withExtension:@"mov"] ;

AVAsset *asset = [AVAsset assetWithURL:assetURL] ;

// Asynchronously load the assets 'tracks' property
NSArray *keys = @[@"tracks"] ;
[asset loadValuesAsynchronouslyForKeys:keys completionHandler:^{
    // Capture the status of the 'tracks' property
    NSError *error = nil;
    AVKeyValueStatus status =
        [asset statusOfValueForKey:@"tracks" error:&error];

    // Switch over the status to determine its state
    switch (status) {
        case AVKeyValueStatusLoaded:
            // Continue Processing
            break;
        case AVKeyValueStatusFailed:
            // Handle failure with error
            break;
        case AVKeyValueStatusCancelled:
            // Handle explicit cancellation
            break;

        default:
            // Handle all other cases
    }
}];
```

```
NSURL *url = // url
AVAsset *asset = [AVAsset assetWithURL:url];

NSArray *keys = @[@"availableMetadataFormats"];
[asset loadValuesAsynchronouslyForKeys:keys completionHandler:^{
    NSMutableArray *metadata = [NSMutableArray array];
    // Collect all metadata for the available formats
    for (NSString *format in asset.availableMetadataFormats) {
        [metadata addObjectsFromArray:[asset metadataForFormat:format]];
    }
    // Process AVMetadataItems
}];
```

```
NSArray *metadata = // Collection of AVMetadataItems

NSString *keySpace = AVMetadataKeySpaceiTunes;
NSString *artistKey = AVMetadataiTunesMetadataKeyArtist;
NSString *albumKey = AVMetadataiTunesMetadataKeyAlbum;

NSArray *artistMetadata = [AVMetadataItem metadataItemsFromArray:metadata
                                                       forKey:artistKey
                                                       keySpace:keySpace];

NSArray *albumMetadata = [AVMetadataItem metadataItemsFromArray:metadata
                                                       forKey:albumKey
                                                       keySpace:keySpace];

AVMetadataItem *artistItem, *albumItem;
if (artistMetadata.count > 0) {
    artistItem = artistMetadata[0];
}

if (albumMetadata.count > 0) {
    albumItem = albumMetadata[0];
}
```

```
NSURL *url = // Song URL
AVAsset *asset = // Asset that has its properties loaded

NSArray *metadata =
[asset metadataForFormat:AVMetadataFormatiTunesMetadata];

for (AVMetadataItem *item in metadata) {
    NSLog(@"%@", item.key, item.value);
}
```

-1452383891: Have A Drink On Me
-1455336876: AC/DC
-1451789708: A. Young - M. Young - B. Johnson
-1453233054: Back In Black
-1453039239: 1980

```
#import "AVMetadataItem+THAdditions.h"

@implementation AVMetadataItem (THAdditions)

- (NSString *)keyString {
    if ([self.key isKindOfClass:[NSString class]]) { // 1
        return (NSString *)self.key;
    }
    else if ([self.key isKindOfClass:[NSNumber class]]) {

        UInt32 keyValue = [(NSNumber *) self.key unsignedIntValue]; // 2

        // Most, but not all, keys are 4 characters. ID3v2.2 keys are
        // only be 3 characters long. Adjust the length if necessary.

        size_t length = sizeof(UInt32); // 3
        if ((keyValue >> 24) == 0) --length;
        if ((keyValue >> 16) == 0) --length;
        if ((keyValue >> 8) == 0) --length;
        if ((keyValue >> 0) == 0) --length;

        long address = (unsigned long)&keyValue;
        address += (sizeof(UInt32) - length);

        // keys are stored in big-endian format, swap
        keyValue = CFSwapInt32BigToHost(keyValue); // 4

        char cstring[length]; // 5
        strncpy(cstring, (char *) address, length);
        cstring[length] = '\0';

        // Replace '@' with '©' to match constants in AVMetadataFormat.h
        if (cstring[0] == '\xA9') { // 6
            cstring[0] = '@';
        }

        return [NSString stringWithCString:(char *) cstring
                           encoding:NSUTF8StringEncoding]; // 7
    }
    else {
        return @"<<unknown>>";
    }
}

@end
```

@nam: Have A Drink On Me

@ART: AC/DC

@wrt: A. Young - M. Young - B. Johnson

@alb: Back In Black

@day: 1980

```
#import <AVFoundation/AVFoundation.h>
#import "THMetadata.h"

typedef void(^THCompletionHandler)(BOOL complete);

@interface THMediaItem : NSObject

@property (copy, readonly) NSString *filename;
@property (copy, readonly) NSString *filetype;
@property (strong, readonly) THMetadata *metadata;
@property (assign, readonly, getter = isEditable) BOOL editable;

- (id)initWithURL:(NSURL *)url;

- (void)prepareWithCompletionHandler:(THCompletionHandler)handler;
- (void)saveWithCompletionHandler:(THCompletionHandler)handler;

@end
```

```
#import "THMediaItem.h"

#import "AVMetadataItem+THAdditions.h"
#import "NSFileManager+DirectoryLocations.h"

#define COMMON_META_KEY      @"commonMetadata"
#define AVAILABLE_META_KEY   @"availableMetadataFormats"

@interface THMediaItem ()
@property (strong) NSURL *url;
@property (strong) AVAsset *asset;
@property (strong) THMetadata *metadata;
@property (strong) NSArray *acceptedFormats;
@property BOOL prepared;
@end

@implementation THMediaItem

- (id)initWithURL:(NSURL *)url {
    self = [super init];
    if (self) {
        _url = url;                                     // 1
        _asset = [AVAsset assetWithURL:url];
        _filename = [url lastPathComponent];
        _filetype = [self fileTypeForURL:url];           // 2
        _editable = ![_filetype isEqualToString:AVFileTypeMPEGLayer3]; // 3
        _acceptedFormats = @[
            AVMetadataFormatQuickTimeMetadata,
```

```
        AVMetadataFormatiTunesMetadata,
        AVMetadataFormatID3Metadata
    ];
}

return self;
}

- (NSString *)fileTypeForURL:(NSURL *)url {
    NSString *ext = [[self.url lastPathComponent] pathExtension];
    NSString *type = nil;
    if ([ext isEqualToString:@"m4a"]) {
        type = AVFileTypeAppleM4A;
    } else if ([ext isEqualToString:@"m4v"]) {
        type = AVFileTypeAppleM4V;
    } else if ([ext isEqualToString:@"mov"]) {
        type = AVFileTypeQuickTimeMovie;
    } else if ([ext isEqualToString:@"mp4"]) {
        type = AVFileTypeMPEG4;
    } else {
        type = AVFileTypeMP3Layer3;
    }
    return type;
}
```

```

- (void)prepareWithCompletionHandler:(THCompletionHandler)completionHandler {
    if (self.prepared) { // 1
        completionHandler(self.prepared);
        return;
    }

    self.metadata = [[THMetadata alloc] init]; // 2

    NSArray *keys = @[COMMON_META_KEY, AVAILABLE_META_KEY];

    [self.asset loadValuesAsynchronouslyForKeys:keys completionHandler:^(

        AVKeyValueStatus commonStatus =
        [self.asset statusOfValueForKey:COMMON_META_KEY error:nil];

        AVKeyValueStatus formatsStatus =
        [self.asset statusOfValueForKey:AVAILABLE_META_KEY error:nil];

        self.prepared = (commonStatus == AVKeyValueStatusLoaded) && // 3
                        (formatsStatus == AVKeyValueStatusLoaded);

        if (self.prepared) {
            for (AVMetadataItem *item in self.asset.commonMetadata) { // 4
                //NSLog(@"%@", item.keyString, item.value);
                [self.metadata addMetadataItem:item forKey:item.commonKey];
            }

            for (id format in self.asset.availableMetadataFormats) { // 5
                if ([self.acceptedFormats containsObject:format]) {
                    NSArray *items = [self.asset metadataForFormat:format];
                    for (AVMetadataItem *item in items) {
                        //NSLog(@"%@", item.keyString, item.value);
                        [self.metadata addMetadataItem:item
                                         forKey:item.keyString];
                    }
                }
            }
        }
    }];

    completionHandler(self.prepared); // 6
}
}

```

```
#import <AVFoundation/AVFoundation.h>

@class THGenre;

@interface THMetadata : NSObject

@property (copy) NSString *name;
@property (copy) NSString *artist;
@property (copy) NSString *albumArtist;
@property (copy) NSString *album;
@property (copy) NSString *grouping;
@property (copy) NSString *composer;
@property (copy) NSString *comments;
@property (strong) NSImage *artwork;
@property (strong) THGenre *genre;

@property NSString *year;
@property NSNumber *bpm;
@property NSNumber *trackNumber;
@property NSNumber *trackCount;
@property NSNumber *discNumber;
@property NSNumber *discCount;

- (void)addMetadataItem:(AVMetadataItem *)item withKey:(id)key;
- (NSArray *)metadataItems;

@end
```

```
#import "THMetadata.h"
#import "THMetadataConverterFactory.h"

@interface THMetadata ()
@property (strong) NSDictionary *keyMapping;
@property (strong) NSMutableDictionary *metadata;
@property (strong) THMetadataConverterFactory *converterFactory;
@end

@implementation THMetadata

- (id)init {
    self = [super init];
    if (self) {
        _keyMapping = [self buildKeyMapping]; // 1
        _metadata = [NSMutableDictionary dictionary]; // 2
        _converterFactory = [[THMetadataConverterFactory alloc] init]; // 3
    }
    return self;
}

- (NSDictionary *)buildKeyMapping {

    return @{
        // Name Mapping
        AVMetadataCommonKeyTitle : THMetadataKeyName,
        // Artist Mapping
        AVMetadataCommonKeyArtist : THMetadataKeyArtist,
        AVMetadataQuickTimeMetadataKeyProducer : THMetadataKeyArtist,
        // Album Artist Mapping
        AVMetadataID3MetadataKeyBand : THMetadataKeyAlbumArtist,
        AVMetadataiTunesMetadataKeyAlbumArtist : THMetadataKeyAlbumArtist,
        @"TP2" : THMetadataKeyAlbumArtist,
        // Continued mappings...
        ...
    };
}
```

```
- (void)addMetadataItem:(AVMetadataItem *)item forKey:(id)key {
    NSString *normalizedKey = self.keyMapping[key]; // 1

    if (normalizedKey) {

        id <THMetadataConverter> converter = // 2
            [self.converterFactory converterForKey:normalizedKey];

        // Extract the value from the AVMetadataItem instance and
        // convert it into a format suitable for presentation.
        id value = [converter displayValueFromMetadataItem:item];

        // Track and Disc numbers/counts are stored as NSDictionary
        if ([value isKindOfClass:[NSDictionary class]]) { // 3
            NSDictionary *data = (NSDictionary *)value;
            for (NSString *currentKey in data) {
                [self setValue:data[currentKey] forKey:currentKey];
            }
        } else {
            [self setValue:value forKey:normalizedKey];
        }

        // Store the AVMetadataItem away for later use
        self.metadata[normalizedKey] = item; // 4
    }
}
```

```
#import <AVFoundation/AVFoundation.h>

@protocol THMetadataConverter <NSObject>

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item;

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
                                         withMetadataItem:(AVMetadataItem *)item;
@end
```

```
#import "THMetadataConverter.h"

@interface ClassName : NSObject <THMetadataConverter>

@end
```

```
# import "THDefaultMetadataConverter.h"

@implementation THDefaultMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {
    return item.value;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
    withMetadataItem:(AVMetadataItem *)item {
    AVMutableMetadataItem *metadataItem = [item mutableCopy];
    metadataItem.value = value;
    return metadataItem;
}

@end
```

```
#import "THArtworkMetadataConverter.h"

@implementation THArtworkMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {
    NSImage *image = nil;
    if ([item.value isKindOfClass:[NSData class]]) { // 1
        image = [[NSImage alloc] initWithData:item.dataValue];
    }
    else if ([item.value isKindOfClass:[NSDictionary class]]) { // 2
        NSDictionary *dict = (NSDictionary *)item.value;
        image = [[NSImage alloc] initWithData:dict[@"data"]];
    }
    return image;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
    withMetadataItem:(AVMetadataItem *)item {
    AVMutableMetadataItem *metadataItem = [item mutableCopy];
    NSImage *image = (NSImage *)value;
    metadataItem.value = image.TIFFRepresentation; // 3
    return metadataItem;
}

@end
```

```
#import "THCommentMetadataConverter.h"

@implementation THCommentMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {

    NSString *value = nil;
    if ([item.value isKindOfClass:[NSString class]]) { // 1
        value = item.stringValue;
    }
    else if ([item.value isKindOfClass:[NSDictionary class]]) { // 2
        NSDictionary *dict = (NSDictionary *)item.value;
        if ([dict[@"identifier"] isEqualToString:@""]) {
            value = dict[@"text"];
        }
    }
    return value;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
    withMetadataItem:(AVMetadataItem *)item {

    AVMutableMetadataItem *metadataItem = [item mutableCopy]; // 3
    metadataItem.value = value;
    return metadataItem;
}

@end
```

```
#import "THTrackMetadataConverter.h"
#import "THMetadataKeys.h"

@implementation THTrackMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {

    NSNumber *number = nil;
    NSNumber *count = nil;

    if ([item.value isKindOfClass:[NSString class]]) { // 1
        NSArray *components =
            [item.stringValue componentsSeparatedByString:@"/"];
        number = @( [components[0] integerValue]);
        count = @( [components[1] integerValue]);
    }
    else if ([item.value isKindOfClass:[NSData class]]) { // 2
        NSData *data = item.dataValue;
        if (data.length == 8) {
            uint16_t *values = (uint16_t *) [data bytes];
            if (values[1] > 0) {
                number = @(CFSwapInt16BigToHost(values[1])); // 3
            }
            if (values[2] > 0) {
                count = @(CFSwapInt16BigToHost(values[2])); // 4
            }
        }
    }
}
```

```

NSMutableDictionary *dict = [NSMutableDictionary dictionaryWithObjects:[NSNull null] forKeys:@[THMetadataKeyTrackNumber, THMetadataKeyTrackCount]];
[dict setObject:[NSNull null] forKey:THMetadataKeyTrackNumber];
[dict setObject:[NSNull null] forKey:THMetadataKeyTrackCount];

return dict;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
    withMetadataItem:(AVMetadataItem *)item {

    AVMutableMetadataItem *metadataItem = [item mutableCopy];

    NSDictionary *trackData = (NSDictionary *)value;
    NSNumber *trackNumber = trackData[THMetadataKeyTrackNumber];
    NSNumber *trackCount = trackData[THMetadataKeyTrackCount];

    uint16_t values[4] = {0}; // 6

    if (trackNumber && !([trackNumber isKindOfClass:[NSNull class]])) {
        values[1] = CFSwapInt16HostToBig([trackNumber unsignedIntValue]); // 7
    }

    if (trackCount && !([trackCount isKindOfClass:[NSNull class]])) {
        values[2] = CFSwapInt16HostToBig([trackCount unsignedIntValue]); // 8
    }

    size_t length = sizeof(values);
    metadataItem.value = [NSData dataWithBytes:&values length:length]; // 9

    return metadataItem;
}

@end

```

```
#import "THDiscMetadataConverter.h"
#import "THMetadataKeys.h"

@implementation THDiscMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {
    NSNumber *number = nil;
    NSNumber *count = nil;

    if ([item.value isKindOfClass:[NSString class]]) { // 1
        NSArray *components =
            [item.stringValue componentsSeparatedByString:@"/"];
        number = @( [components[0] integerValue]);
        count = @( [components[1] integerValue]);
    }
    else if ([item.value isKindOfClass:[NSData class]]) { // 2
        NSData *data = item.dataValue;
        if (data.length == 6) {
            uint16_t *values = (uint16_t *) [data bytes];
            if (values[1] > 0) {
                number = @(CFSwapInt16BigToHost(values[1])); // 3
            }
            if (values[2] > 0) {
                count = @(CFSwapInt16BigToHost(values[2])); // 4
            }
        }
    }
}

NSMutableDictionary *dict = [NSMutableDictionary dictionary]; // 5
[dict setObject:number ?: [NSNull null] forKey:THMetadataKeyDiscNumber];
[dict setObject:count ?: [NSNull null] forKey:THMetadataKeyDiscCount];

return dict;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
    withMetadataItem:(AVMetadataItem *)item {
    AVMutableMetadataItem *metadataItem = [item mutableCopy];

    NSDictionary *discData = (NSDictionary *)value;
    NSNumber *discNumber = discData[THMetadataKeyDiscNumber];
    NSNumber *discCount = discData[THMetadataKeyDiscCount];

    uint16_t values[3] = {0}; // 6
```

```
if (discNumber && (![discNumber isKindOfClass:[NSNull class]])) {
    values[1] = CFSwapInt16HostToBig([discNumber unsignedIntValue]); // 7
}

if (discCount && (![discCount isKindOfClass:[NSNull class]])) {
    values[2] = CFSwapInt16HostToBig([discCount unsignedIntValue]); // 8
}

size_t length = sizeof(values);
metadataItem.value = [NSData dataWithBytes:values length:length]; // 9

return metadataItem;
}

@end
```

```
#import "THGenreMetadataConverter.h"
#import "THGenre.h"

@implementation THGenreMetadataConverter

- (id)displayValueFromMetadataItem:(AVMetadataItem *)item {
    THGenre *genre = nil;

    if ([item.value isKindOfClass:[NSString class]]) { // 1
        if ([item.keySpace isEqualToString:AVMetadataKeySpaceID3]) {
            // ID3v2.4 stores the genre as an index value
            if (item.numberValue) { // 2
                NSUInteger genreIndex = [item.numberValue unsignedIntValue];
                genre = [THGenre id3GenreWithIndex:genreIndex];
            } else {
                genre = [THGenre id3GenreWithName:item.stringValue]; // 3
            }
        } else {
            genre = [THGenre videoGenreWithName:item.stringValue]; // 4
        }
    }
    else if ([item.value isKindOfClass:[NSData class]]) { // 5
        NSData *data = item.dataValue;
        if (data.length == 2) {
            uint16_t *values = (uint16_t *)[data bytes];
            uint16_t genreIndex = CFSwapInt16BigToHost(values[0]);
            genre = [THGenre iTunesGenreWithIndex:genreIndex];
        }
    }
}
```

```

        }
    }
    return genre;
}

- (AVMetadataItem *)metadataItemFromDisplayValue:(id)value
    withMetadataItem:(AVMetadataItem *)item {

    AVMutableMetadataItem *metadataItem = [item mutableCopy];

    THGenre *genre = (THGenre *)value;

    if ([item.value isKindOfClass:[NSString class]]) { // 6
        metadataItem.value = genre.name;
    }
    else if ([item.value isKindOfClass:[NSData class]]) { // 7
        NSData *data = item.dataValue;
        if (data.length == 2) {
            uint16_t value = CFSwapInt16HostToBig(genre.index + 1);
            size_t length = sizeof(value);
            metadataItem.value = [NSData dataWithBytes:&value length:length];
        }
    }

    return metadataItem;;
}

```

@end

```
- (NSArray *)metadataItems {

    NSMutableArray *items = [NSMutableArray array]; // 1

    // Add track number/count if applicable
    [self addMetadataItemForNumber:self.trackNumber
                           count:self.trackCount
                         numberKey:kTHMetadataKeyTrackNumber
                         countKey:kTHMetadataKeyTrackCount
                           toArray:items];
    // 2

    // Add disc number/count if applicable
    [self addMetadataItemForNumber:self.discNumber
                           count:self.discCount
                         numberKey:kTHMetadataKeyDiscNumber
                         countKey:kTHMetadataKeyDiscCount
                           toArray:items];
    // 3

    NSMutableDictionary *metaDict = [self.metadata mutableCopy]; // 4
    [metaDict removeObjectForKey:kTHMetadataKeyTrackNumber];
    [metaDict removeObjectForKey:kTHMetadataKeyDiscNumber];

    for (NSString *key in metaDict) {
        id <THMetadataConverter> converter =
            [self.converterFactory converterForKey:key];
        // 5
    }
}
```

```
    id value = [self valueForKey:key]; // 6

    AVMetadataItem *item =
        [converter metadataItemFromDisplayValue:value
                           withMetadataItem:metaDict[key]];
    if (item) {
        [items addObject:item];
    }
}

return items;
}

- (void)addMetadataItemForNumber:(NSNumber *)number
                           count:(NSNumber *)count
                         numberKey:(NSString *)numberKey
                        countKey:(NSString *)countKey
                      toArray:(NSMutableArray *)items {

    id <THMetadataConverter> converter =
        [self.converterFactory converterForKey:numberKey];

    NSDictionary *data = @{@"numberKey" : number ?: [NSNull null], // 3
                           "countKey" : count ?: [NSNull null]};

    AVMetadataItem *sourceItem = self.metadata[numberKey];

    AVMetadataItem *item = // 4
        [converter metadataItemFromDisplayValue:data
                           withMetadataItem:sourceItem];
    if (item) {
        [items addObject:item];
    }
}
```

```
- (void)saveWithCompletionHandler:(THCompletionHandler)handler {

    NSString *presetName = AVAssetExportPresetPassthrough; // 1
    AVAssetExportSession *session =
        [[AVAssetExportSession alloc] initWithAsset:self.asset
                                         presetName:presetName];

    NSURL *outputURL = [self tempURL]; // 2
    session.outputURL = outputURL;
    session.outputFileType = self.filetype;
    session.metadata = [self.metadata metadataItems]; // 3

    [session exportAsynchronouslyWithCompletionHandler:^{
        AVAssetExportSessionStatus status = session.status;
        BOOL success = (status == AVAssetExportSessionStatusCompleted);
        if (success) { // 4
            NSURL *sourceURL = self.url;
            NSFileManager *manager = [NSFileManager defaultManager];
            [manager removeItemAtURL:sourceURL error:nil];
            [manager moveItemAtURL:outputURL toURL:sourceURL error:nil];
            [self reset]; // 5
        }
        if (handler) {
            handler(success);
        }
    }];
}

- (NSURL *)tempURL {
    NSString *tempDir = NSTemporaryDirectory();
    NSString *ext = [[self.url lastPathComponent] pathExtension];
    NSString *tempName = [NSString stringWithFormat:@"temp.%@", ext];
    NSString *tempPath = [tempDir stringByAppendingPathComponent:tempName];
    return [NSURL fileURLWithPath:tempPath];
}

- (void)reset {
    _prepared = NO;
    _asset = [AVAsset assetWithURL:self.url];
}
```

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // 1. Define the asset URL
    NSURL *assetURL =
        [[NSBundle mainBundle] URLForResource:@"waves" withExtension:@"mp4"];

    // 2. Create an instance of AVAsset
    AVAsset *asset = [AVAsset assetWithURL:assetURL];

    // 3. Create an AVPlayerItem with a pointer to the asset to play
    AVPlayerItem *playerItem = [AVPlayerItem playerItemWithAsset:asset];

    // 4. Create an instance of AVPlayer with a pointer to the player item
    self.player = [AVPlayer playerWithPlayerItem:playerItem];

    // 5. Create a player layer to direct the video content
    AVPlayerLayer *playerLayer =
        [AVPlayerLayer playerLayerWithPlayer:self.player];

    // 6. Attach layer into layer hierarchy
    [self.view.layer addSublayer:playerLayer];
}
```

```
static const NSString *PlayerItemStatusContext;

- (void)viewDidLoad {

    ...

    AVPlayerItem *playerItem = [AVPlayerItem playerItemWithAsset:asset];

    [playerItem addObserver:self
                    forKeyPath:@"status"
                      options:0
                     context:&PlayerItemStatusContext];

    self.player = [AVPlayer playerWithPlayerItem:playerItem];
}

- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {

    if (context == &PlayerItemStatusContext) {

        AVPlayerItem *playerItem = (AVPlayerItem *)object;
        if (playerItem.status == AVPlayerItemStatusReadyToPlay) {
            // proceed with playback
        }
    }
}
```

```
typedef struct {
    CMTimeValue value;
    CMTimeScale timescale;
    CMTimeFlags flags;
    CMTimeEpoch epoch;
} CMTime;
```

```
// 0.5 seconds
CMTime halfSecond = CMTimeMake(1, 2);

// 5 seconds
CMTime fiveSeconds = CMTimeMake(5, 1);

// One sample from a 44.1 kHz audio file
CMTime oneSample = CMTimeMake(1, 44100);

// Zero time value
CMTime zeroTime = kCMTimeZero;
```

```
#import "THTransport.h"

@class AVPlayer;

@interface THPlayerView : UIView

- (id)initWithPlayer:(AVPlayer *)player;

@property (nonatomic, readonly) id <THTransport> transport;

@end
```

```
#import "THPlayerView.h"
#import "THOverlayView.h"
#import <AVFoundation/AVFoundation.h>

@interface THPlayerView ()
@property (strong, nonatomic) THOverlayView *overlayView; // 1
@end

@implementation THPlayerView

+ (Class)layerClass { // 2
    return [AVPlayerLayer class];
}

- (id)initWithPlayer:(AVPlayer *)player {
    self = [super initWithFrame:CGRectMakeZero]; // 3
    if (self) {
        self.backgroundColor = [UIColor blackColor];
        self.autoresizingMask = UIViewAutoresizingFlexibleHeight |
                               UIViewAutoresizingFlexibleWidth;

        [(AVPlayerLayer *) [self layer] setPlayer:player]; // 4

        [[[NSBundle mainBundle] loadNibNamed:@"THOverlayView"
                                         owner:self
                                         options:nil];

        [self addSubview:_overlayView];
    }
    return self;
}

- (void)layoutSubviews {
    [super layoutSubviews];
    self.overlayView.frame = self.bounds;
}

- (id <THTransport>)transport {
    return self.overlayView;
}

@end
```

```
@interface THPlayerController : NSObject

- (id)initWithURL:(NSURL *)assetURL;

@property (strong, nonatomic, readonly) UIView *view;

@end
```

```
#import "THPlayerController.h"
#import <AVFoundation/AVFoundation.h>
#import "THTransport.h"
#import "THPlayerView.h"
#import "AVAsset+THAdditions.h"
#import "UIAlertView+THAdditions.h"

// AVPlayerItem's status property
#define STATUS_KEYPATH @"status"

// Refresh interval for timed observations of AVPlayer
#define REFRESH_INTERVAL 0.5f

// Define this constant for the key-value observation context.
static const NSString *PlayerItemStatusContext;

@interface THPlayerController () <THTransportDelegate>

@property (strong, nonatomic) AVAsset *asset;
@property (strong, nonatomic) AVPlayerItem *playerItem;
@property (strong, nonatomic) AVPlayer *player;
@property (strong, nonatomic) THPlayerView *playerView;

@property (weak, nonatomic) id <THTransport> transport;

@property (strong, nonatomic) id timeObserver;
@property (strong, nonatomic) id itemEndObserver;
@property (assign, nonatomic) float lastPlaybackRate;

@end
```

```
@protocol THTransportDelegate <NSObject>

- (void)play;
- (void)pause;
- (void)stop;

- (void)scrubbingDidStart;
- (void)scrubbedToTime:(NSTimeInterval)time;
- (void)scrubbingDidEnd;

- (void)jumpedToTime:(NSTimeInterval)time;

@end

@protocol THTransport <NSObject>

@property (weak, nonatomic) id <THTransportDelegate> delegate;

- (void)setTitle:(NSString *)title;
- (void)setCurrentTime:(NSTimeInterval)time duration:(NSTimeInterval)duration;
- (void)setScrubbingTime:(NSTimeInterval)time;
- (void)playbackComplete;

@end
```

```
@implementation THPlayerController

#pragma mark - Setup

- (id)initWithURL:(NSURL *)assetURL {
    self = [super init];
    if (self) {
        _asset = [AVAsset assetWithURL:assetURL]; // 1
        [self prepareToPlay];
    }
    return self;
}

- (void)prepareToPlay {
    NSArray *keys = @[@"tracks", @"duration", @"commonMetadata"];
    self.playerItem = [AVPlayerItem playerItemWithAsset:self.asset // 2
                      automaticallyLoadedAssetKeys:keys];

    [self.playerItem addObserver:self // 3
                          forKeyPath:STATUS_KEYPATH
                            options:0
                          context:&PlayerItemStatusContext];

    self.player = [AVPlayer playerWithPlayerItem:self.playerItem]; // 4

    self.playerView = [[THPlayerView alloc] initWithPlayer:self.player]; // 5
    self.transport = self.playerView.transport;
    self.transport.delegate = self;
}

// More methods to follow ...

@end
```

```
- (void)observeValueForKeyPath:(NSString *)keyPath
    withObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {

    if (context == &PlayerItemStatusContext) {

        dispatch_async(dispatch_get_main_queue(), ^{
            // 1
            [self.playerItem removeObserver:self forKeyPath:STATUS_KEYPATH];

            if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {

                // Set up time observers. // 2
                [self addPlayerItemTimeObserver];
                [self addItemEndObserverForPlayerItem];

                CMTime duration = self.playerItem.duration;

                // Synchronize the time display // 3
                [self.transport setCurrentTime:CMTimeGetSeconds(kCMTimeZero)
                    duration:CMTimeGetSeconds(duration)];

                // Set the video title.
                [self.transport setTitle:self.asset.title]; // 4

                [self.player play];
                // 5
            } else {
                UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"Error"
                                                                message:@"Failed to load video"];
            }
        });
    }
}
```

```
- (void)addItemEndObserverForPlayerItem {

    NSString *name = AVPlayerItemDidPlayToEndTimeNotification;

    NSOperationQueue *queue = [NSOperationQueue mainQueue];

    __weak THPlayerController *weakSelf = self; // 1
    void (^callback)(NSNotification *note) = ^(NSNotification *notification) {
        [weakSelf.player seekToTime:kCMTimeZero // 2
            completionHandler:^(BOOL finished) {
                [weakSelf.transport playbackComplete]; // 3
            }];
    };

    self.itemEndObserver = // 4
        [[NSNotificationCenter defaultCenter] addObserverForName:name
                                                 object:self.playerItem
                                                 queue:queue
                                               usingBlock:callback];
}

- (void)dealloc {
    if (self.itemEndObserver) { // 5
        NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
        [nc removeObserver:self.itemEndObserver
                      name:AVPlayerItemDidPlayToEndTimeNotification
                      object:self.player.currentItem];
        self.itemEndObserver = nil;
    }
}
```

```
- (void)play {
    [self.player play];
}

- (void)pause {
    self.lastPlaybackRate = self.player.rate;
    [self.player pause];
}

- (void)stop {
    [self.player setRate:0.0f];
    [self.transport playbackComplete];
}

- (void)jumpedToTime:(NSTimeInterval)time {
    [self.player seekToTime:CMTIMEMakeWithSeconds(time, NSEC_PER_SEC)];
}
```

```
- (void)scrubbingDidStart { // 1
    self.lastPlaybackRate = self.player.rate;
    [self.player pause];
    [self.player removeTimeObserver:self.timeObserver];
}

- (void)scrubbedToTime:(NSTimeInterval)time { // 2
    [self.playerItem cancelPendingSeeks];
    [self.player seekToTime:CMTIMEMakeWithSeconds(time, NSEC_PER_SEC)];
}

- (void)scrubbingDidEnd { // 3
    [self addPlayerItemTimeObserver];
    if (self.lastPlaybackRate > 0.0f) {
        [self.player play];
    }
}
```

```
#import "THPlayerController.h"
#import <AVFoundation/AVFoundation.h>
#import "THTransport.h"
#import "THPlayerView.h"
#import "AVAsset+THAdditions.h"
#import "UIAlertView+THAdditions.h"
#import "THThumbnail.h"

...
@interface THPlayerController () <THTransportDelegate>

@property (strong, nonatomic) AVAsset *asset;
@property (strong, nonatomic) AVPlayerItem *playerItem;
@property (strong, nonatomic) AVPlayer *player;
@property (strong, nonatomic) THPlayerView *playerView;

@property (weak, nonatomic) id <THTransport> transport;

@property (strong, nonatomic) id timeObserver;
@property (strong, nonatomic) id itemEndObserver;
@property (assign, nonatomic) float lastPlaybackRate;

@property (strong, nonatomic) AVAssetImageGenerator *imageGenerator;
@end
```

```
- (void)observeValueForKeyPath:(NSString *)keyPath
    withObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {

    if (context == &PlayerItemStatusContext) {

        dispatch_async(dispatch_get_main_queue(), ^{
            [self.playerItem removeObserver:self forKeyPath:STATUS_KEYPATH];

            if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {

                // Set up time observers.
                [self addPlayerItemTimeObserver];
                [self addItemEndObserverForPlayerItem];

                CMTime duration = self.playerItem.duration;

                // Synchronize the time display
                [self.transport setCurrentTime:CMTimeGetSeconds(kCMTimeZero)
                    duration:CMTimeGetSeconds(duration)];

                // Set the video title.
                [self.transport setTitle:self.asset.title];

                [self.player play];

                [self generateThumbnails];
            } else {
                UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"Error"
                                                                message:@"Failed to load video"];
                [alertView show];
            }
        });
    }
}

- (void)generateThumbnails {
```

```
- (void)generateThumbnails {

    self.imageGenerator = // 1
        [AVAssetImageGenerator assetImageGeneratorWithAsset:self.asset];

    // Generate the @2x equivalent
    self.imageGenerator.maximumSize = CGSizeMake(200.0f, 0.0f); // 2

    CMTime duration = self.asset.duration;

    NSMutableArray *times = [NSMutableArray array]; // 3
    CMTimeValue increment = duration.value / 20;
    CMTimeValue currentValue = kCMTimeZero;
    while (currentValue <= duration.value) {
        CMTime time = CMTimeMake(currentValue, duration.timescale);
        [times addObject:[NSValue valueWithCMTime:time]];
        currentValue += increment;
    }

    __block NSUInteger imageCount = times.count; // 4
    __block NSMutableArray *images = [NSMutableArray array];

    AVAssetImageGeneratorCompletionHandler handler; // 5

    handler = ^(CMTime requestedTime,
                CGImageRef imageRef,
                CMTime actualTime,
                AVAssetImageGeneratorResult result,
                NSError *error) {
```

```
if (result == AVAssetImageGeneratorSucceeded) { // 6
    UIImage *image = [UIImage imageWithCGImage:imageRef];
    id thumbnail =
        [THThumbnail thumbnailWithImage:image time:actualTime];
    [images addObject:thumbnail];
} else {
    NSLog(@"Failed to create thumbnail image.");
}

// If the decremented image count is at 0, we're all done.
if (--imageCount == 0) { // 7
    dispatch_async(dispatch_get_main_queue(), ^{
        NSString *name = THThumbnailsGeneratedNotification;
        NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
        [nc postNotificationName:name object:images];
    });
}
};

[self.imageGenerator generateCGImagesAsynchronouslyForTimes:times // 8
completionHandler:handler];
}
```

```
NSArray *mediaCharacteristics =
    self.asset.availableMediaCharacteristicsWithMediaSelectionOptions;
for (NSString *characteristic in mediaCharacteristics) {
    AVMediaSelectionGroup *group =
        [self.asset mediaSelectionGroupForMediaCharacteristic:characteristic];
    NSLog(@"%@", characteristic);
    for (AVMediaSelectionOption *option in group.options) {
        NSLog(@"Option: %@", option.displayName);
    }
}
```

[AVMediaCharacteristicLegible]

Option: English

Option: Italian

Option: Portuguese

Option: Russian

[AVMediaCharacteristicAudible]

Option: English

```
AVMediaSelectionGroup *group =
    [self.asset mediaSelectionGroupForMediaCharacteristic:characteristic];
NSLocale *russianLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"ru_RU"];
NSArray *options =
    [AVMediaSelectionGroup mediaSelectionOptionsFromArray:group.options
                                                withLocale:russianLocale];
AVMediaSelectionOption *option = [options firstObject];
[self.playerItem selectMediaOption:option inMediaSelectionGroup:group];
```

```
- (void)prepareToPlay {
    NSArray *keys = @[
        @"tracks",
        @"duration",
        @"commonMetadata",
        @"availableMediaCharacteristicsWithMediaSelectionOptions"
    ];
    ...
}

- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {

    if (context == &PlayerItemStatusContext) {
        dispatch_async(dispatch_get_main_queue(), ^{
            if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {
                ...
                [self loadMediaOptions];
            } else {
                UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"Error"
                                                                message:@"Failed to load video"];
                [alertView show];
            }
        });
    }
}

- (void)loadMediaOptions {
}
```

```
- (void)loadMediaOptions {
    NSString *mc = AVMediaCharacteristicLegible; // 1
    AVMediaSelectionGroup *group =
        [self.asset mediaSelectionGroupForMediaCharacteristic:mc]; // 2
    if (group) {
        NSMutableArray *subtitles = [NSMutableArray array]; // 3
        for (AVMediaSelectionOption *option in group.options) {
            [subtitles addObject:option.displayName];
        }
        [self.transport setSubtitles:subtitles];
    } // 4
    else {
        [self.transport setSubtitles:nil];
    }
}
```

```
- (void)subtitleSelected:(NSString *)subtitle {
    NSString *mc = AVMediaCharacteristicLegible;
    AVMediaSelectionGroup *group =
        [self.asset mediaSelectionGroupForMediaCharacteristic:mc];           // 1
    BOOL selected = NO;
    for (AVMediaSelectionOption *option in group.options) {
        if ([option.displayName isEqualToString:subtitle]) {
            [self.playerItem selectMediaOption:option
                inMediaSelectionGroup:group];                                // 2
            selected = YES;
        }
    }
    if (!selected) {
        [self.playerItem selectMediaOption:nil
            inMediaSelectionGroup:group];                                // 3
    }
}
```

```
CGRect rect = // desired frame  
MPVolumeView *volumeView = [[MPVolumeView alloc] initWithFrame:rect];  
[self.view addSubview:volumeView];
```

```
MPVolumeView *volumeView = [ [MPVolumeView alloc] init];
volumeView.showsVolumeSlider = NO;
[volumeView sizeToFit];
[transportView addSubview:volumeView];
```

```
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    NSURL *url =
    [[NSBundle mainBundle] URLForResource:@"video" withExtension:@"m4v"] ;

    AVPlayerViewController *controller = [[AVPlayerViewController alloc] init];
    controller.player = [AVPlayer playerWithURL:url];

    self.window.rootViewController = controller;

    return YES;
}

@end
```

```
#import "THDocument.h"
#import <AVFoundation/AVFoundation.h>
#import <AVKit/AVKit.h>

@interface THDocument ()
@property (weak) IBOutlet AVPlayerView *playerView;
@end

@implementation THDocument

- (void>windowControllerDidLoadNib:(NSWindowController *)aController {
    [super windowControllerDidLoadNib:aController];
}

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

- (BOOL)readFromURL:(NSURL *)url
              ofType:(NSString *)typeName
            error:(NSError *__autoreleasing *)outError {
    return YES;
}

@end
```

```
self.playerView.player = [AVPlayer playerWithURL:[self fileURL]];  
self.playerView.showsSharingServiceButton = YES;
```

```
#import "THDocument.h"
#import <AVFoundation/AVFoundation.h>
#import <AVKit/AVKit.h>
#define STATUS_KEY @"status"

@interface THDocument ()

@property (strong) AVAsset *asset; // 1
@property (strong) AVPlayerItem *playerItem;
@property (strong) NSArray *chapters;

@property (weak) IBOutlet AVPlayerView *playerView;

@end

@implementation THDocument

- (void>windowControllerDidLoadNib:(NSWindowController *)controller {
    [super windowControllerDidLoadNib:controller];

    [self setupPlaybackStackWithURL:[self fileURL]]; // 2
}

- (void)setupPlaybackStackWithURL:(NSURL *)url {
    self.asset = [AVAsset assetWithURL:url];

    NSArray *keys = @[@"commonMetadata", @"availableChapterLocales"]; // 3

    self.playerItem = [AVPlayerItem playerItemWithAsset:self.asset
                                                automaticallyLoadedAssetKeys:keys]; // 4

    [self.playerItem addObserver:self
                           forKeyPath:STATUS_KEY
                             options:0 context:NULL]; // 5

    self.playerView.player = [AVPlayer playerWithPlayerItem:self.playerItem];
    self.playerView.showsSharingServiceButton = YES;
}

- (void)observeValueForKeyPath:(NSString *)keyPath
                       ofObject:(id)object
                         change:(NSDictionary *)change
                        context:(void *)context {

    if ([keyPath isEqualToString:STATUS_KEY]) {
```

```
if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {
    NSString *title = [self titleForAsset:self.asset]; // 6
    if (title) {
        self.windowForSheet.title = title;
    }
    self.chapters = [self chaptersForAsset:self.asset]; // 7
}

[self.playerItem removeObserver:self forKeyPath:STATUS_KEY]; // 8

}

}

- (NSString *)titleForAsset:(AVAsset *)asset {
    return nil;
}

- (NSArray *)chaptersForAsset:(AVAsset *)asset {
    return nil;
}
```

```
- (NSString *)titleInMetadata:(NSArray *)metadata {
    NSArray *items = // 1
        [AVMetadataItem metadataItemsFromArray:metadata
            forKey:AVMetadataCommonKeyTitle
            keySpace:AVMetadataKeySpaceCommon];

    return [[items firstObject] stringValue]; // 2
}

- (NSString *)titleForAsset:(AVAsset *)asset {
    NSString *title = [self titleInMetadata:asset.commonMetadata]; // 3
    if (title && (![title isEqualToString:@""])) {
        return title;
    }
    return nil;
}
```

chapterMetadataGroupsWithTitleLocale:containingItemsWithCommonKeys:
chapterMetadataGroupsBestMatchingPreferredLanguages:

```
NSURL *url = // asset URL;
AVAsset *asset = [AVAsset assetWithURL:url];
NSString *key = @"availableChapterLocales";

[asset loadValuesAsynchronouslyForKeys:@[key] completionHandler:^{
    AVKeyValueStatus status = [asset statusOfValueForKey:key error:nil];

    if (status == AVKeyValueStatusLoaded) {
        NSArray *langs = [NSLocale preferredLanguages];
        NSArray *chapterMetadata =
            [asset chapterMetadataGroupsBestMatchingPreferredLanguages:langs];

        // Process AVTimeMetadataGroup objects
    }
}];
```

```
- (NSArray *)chaptersForAsset:(AVAsset *)asset {

    NSArray *languages = [NSLocale preferredLanguages]; // 1

    NSArray *metadataGroups = // 2
        [asset chapterMetadataGroupsBestMatchingPreferredLanguages:languages];

    NSMutableArray *chapters = [NSMutableArray array];

    for (NSUInteger i = 0; i < metadataGroups.count; i++) {
        AVTimedMetadataGroup *group = metadataGroups[i];

        CMTime time = group.timeRange.start;
        NSUInteger number = i + 1;
        NSString *title = [self titleInMetadata:group.items];

        THChapter *chapter = // 3
            [THChapter chapterWithTime:time number:number title:title];

        [chapters addObject:chapter];
    }
    return chapters;
}
```

```
- (void)setupActionMenu {
    NSMenu *menu = [[NSMenu alloc] init]; // 1
    [menu addItem:[[NSMenuItem alloc] initWithTitle:@"Previous Chapter"
                                                action:@selector(previousChapter:)
                                              keyEquivalent:@""]];
    [menu addItem:[[NSMenuItem alloc] initWithTitle:@"Next Chapter"
                                                action:@selector(nextChapter:)
                                              keyEquivalent:@""]];
    self.playerView.actionPopUpButtonMenu = menu; // 2
}

- (void)previousChapter:(id)sender {

}

- (void)nextChapter:(id)sender {

}
```

```
- (void)observeValueForKeyPath:(NSString *)keyPath
    withObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {
    if (self.playerItem.status == AVPlayerItemStatusReadyToPlay) {
        NSString *title = [self titleForAsset:self.asset];
        if (title) {
            self.windowForSheet.title = title;
        }
        self.chapters = [self chaptersForAsset:self.asset];

        // Create action menu if chapters are available
        if ([self.chapters count] > 0) {
            [self setupActionMenu];
        }
    }

    [self.playerItem removeObserver:self forKeyPath:STATUS_KEY];
}
```

```

- (THChapter *)findPreviousChapter {
    CMTIME playerTime = self.playerItem.currentTime;
    CMTIME currentTime = CMTIMESubtract(playerTime, CMTIMEMake(3, 1)); // 1
    CMTIME pastTime = kCMTIMENegativeInfinity;

    CMTIMERange timeRange = CMTIMERangeMake(pastTime, currentTime); // 2

    return [self findChapterInTimeRange:timeRange reverse:YES]; // 3
}

- (THChapter *)findNextChapter {
    CMTIME currentTime = self.playerItem.currentTime; // 4
    CMTIME futureTime = kCMTIMEPositiveInfinity;

    CMTIMERange timeRange = CMTIMERangeMake(currentTime, futureTime); // 5

    return [self findChapterInTimeRange:timeRange reverse:NO]; // 6
}

- (THChapter *)findChapterInTimeRange:(CMTIMERange)timeRange
    reverse:(BOOL)reverse {
    __block THChapter *matchingChapter = nil;

    NSEnumerationOptions options = reverse ? NSEnumerationReverse : 0; // 7
    [self.chapters enumerateObjectsWithOptions:options
        usingBlock:^(id obj,
                     NSUInteger idx,
                     BOOL *stop) {

        if ([(THChapter *)obj isInTimeRange:timeRange]) { // 8
            matchingChapter = obj;
            *stop = YES;
        }
    }];
    return matchingChapter;
}

```

```
- (BOOL)isInTimeRange:(CMTimeRange)timeRange {
    return CMTIME_COMPARE_INLINE(_time, >, timeRange.start) &&
           CMTIME_COMPARE_INLINE(_time, <, timeRange.duration);
}
```

```
- (void)previousChapter:(id)sender {
    [self skipToChapter:[self findPreviousChapter]]; // 1
}

- (void)nextChapter:(id)sender {
    [self skipToChapter:[self findNextChapter]]; // 2
}

- (void)skipToChapter:(THChapter *)chapter { // 3
    [self.playerItem seekToTime:chapter.time completionHandler:^(BOOL done) {
        [self.playerView flashChapterNumber:chapter.number
                                         chapterTitle:chapter.title];
    }];
}
```

```
- (IBAction)startTrimming:(id)sender {
    [self.playerView beginTrimmingWithCompletionHandler:NULL];
}
```

```
- (BOOL)validateUserInterfaceItem:(id <NSValidatedUserInterfaceItem>)item {
    SEL action = [item action];
    if (action == @selector(startTrimming:)) {
        return self.playerView.canBeginTrimming;
    }
    return YES;
}
```

```
#import "THEExportWindowController.h"

@interface THDocument () <THEExportWindowControllerDelegate>

@property (strong) AVAsset *asset;
@property (strong) AVPlayerItem *playerItem;
@property (strong) NSArray *chapters;
@property (strong) AVAssetExportSession *exportSession;
@property (strong) THEExportWindowController *exportController;

@property (weak) IBOutlet AVPlayerView *playerView;

@end
```

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

    [self.playerView.player pause];                                // 1

    NSSavePanel *savePanel = [NSSavePanel savePanel];

    [savePanel beginSheetModalForWindow:self.windowForSheet
        completionHandler:^(NSInteger result) {

        if (result == NSFileHandlingPanelOKButton) {
            // Order out save panel as the export window will be shown
            [savePanel orderOut:nil];

            NSString *preset = AVAssetExportPresetAppleM4V720pHD;
            self.exportSession =                                         // 2
                [[AVAssetExportSession alloc] initWithAsset:self.asset
                    presetName:preset];

            CMTime startTime = self.playerItem.reversePlaybackEndTime;
            CMTime endTime = self.playerItem.forwardPlaybackEndTime;
            CMTimeRange timeRange = CMTimeRangeMake(startTime, endTime); // 3

            // Configure the export session                               // 4
            self.exportSession.timeRange = timeRange;
            self.exportSession.outputFileType =
                [self.exportSession.supportedFileTypes firstObject];
            self.exportSession.outputURL = savePanel.URL;
        }
    }];
}
```

```
    self.exportController = [[THEExportWindowController alloc] init];
    self.exportController.exportSession = self.exportSession;
    self.exportController.delegate = self;
    [self.windowForSheet beginSheet:self.exportController.window // 5
                           completionHandler:nil];

    [self.exportSession exportAsynchronouslyWithCompletionHandler:^{
        // Tear down // 6
        [self.windowForSheet endSheet:self.exportController.window];
        self.exportController = nil;
        self.exportSession = nil;
    }];
}

}];

- (void)exportDidCancel {
    [self.exportSession cancelExport]; // 7
}
```

```
#import "THDocument.h"
#import <AVFoundation/AVFoundation.h>
#import <AVKit/AVKit.h>
#import "THChapter.h"
#import <QTKit/QTKit.h>
#import "NSFileManager+THAdditions.h"
#import "THWindow.h"

#import "THEExportWindowController.h"

#define STATUS_KEY @"status"

@interface THDocument () <THEExportWindowControllerDelegate>

@property (strong) AVAsset *asset;
@property (strong) AVPlayerItem *playerItem;
@property (strong) NSArray *chapters;
@property (strong) AVAssetExportSession *exportSession;
@property (strong) THEExportWindowController *exportController;

@property BOOL modernizing;

@property (weak) IBOutlet AVPlayerView *playerView;

@end

@implementation THDocument

#pragma mark - NSDocument Methods

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

- (void>windowControllerDidLoadNib:(NSWindowController *)controller {
    [super windowControllerDidLoadNib:controller];

    if (!self.modernizing) {
        [self setupPlaybackStackWithURL:[self fileURL]];
    } else {
        [(id)controller.window showConvertingView];
    }
}
```

```
- (BOOL)readFromURL:(NSURL *)url
              ofType:(NSString *)typeName
                error:(NSError *__autoreleasing *)outError {

    NSError *error = nil;

    if ([QTMovieModernizer requiresModernization:url error:&error]) {           // 1

        self.modernizing = YES;

        NSURL *destURL = [self tempURLForURL:url];                                // 2

        if (!destURL) {
            self.modernizing = NO;
            NSLog(@"Error creating destination URL, skipping modernization.");
            return NO;
        }

        QTMovieModernizer *modernizer =                                         // 5
            [[QTMovieModernizer alloc] initWithSourceURL:url
                                         destinationURL:destURL];

        modernizer.outputFormat = QTMovieModernizerOutputFormat_H264;           // 6
```

```
[modernizer modernizeWithCompletionHandler:^{
    if (modernizer.status ==
        QTMovieModernizerStatusCompletedWithSuccess) { // 7

        dispatch_async(dispatch_get_main_queue(), ^{
            [self setupPlaybackStackWithURL:destURL]; // 8
            [(id)self.windowForSheet hideConvertingView];
        });
    }
}];

return YES;
}

- (NSURL *)tempURLForURL:(NSURL *)url {

    NSFileManager *fileManager = [NSFileManager defaultManager];
    NSString *dirPath = // 3
        [fileManager temporaryDirectoryWithTemplateString:@"kittime.XXXXXX"];

    if (dirPath) { // 4
        NSString *filePath =
            [dirPath stringByAppendingPathComponent:[url lastPathComponent]];
        return [NSURL fileURLWithPath:filePath];
    }

    return nil;
}
```

```
AVCaptureDevice *videoDevice =  
[AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];
```

```
NSError *error;  
  
AVCaptureDeviceInput *videoInput =  
    [AVCaptureDeviceInput deviceInputWithDevice:videoDevice error:&error];
```

```
// 1. Create a capture session.  
AVCaptureSession *session = [[AVCaptureSession alloc] init];  
  
// 2. Get a reference to the default camera.  
AVCaptureDevice *cameraDevice =  
    [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];  
  
// 3. Create a device input for the camera.  
NSError *error;  
AVCaptureDeviceInput *cameraInput =  
    [AVCaptureDeviceInput deviceInputWithDevice:cameraDevice error:&error];  
  
// 4. Connect the input to the session.  
if ([session canAddInput:cameraInput]) {  
    [session addInput:cameraInput];  
}  
  
// 5. Create an AVCaptureOutput to capture still images.  
AVCaptureStillImageOutput *imageOutput =  
    [[AVCaptureStillImageOutput alloc] init];  
  
// 6. Add the output to the session.  
if ([session canAddOutput:imageOutput]) {  
    [session addOutput:imageOutput];  
}  
  
// 7. Start the session and begin the flow of data.  
[session startRunning];
```

```
#import <AVFoundation/AVFoundation.h>

@protocol THPreviewViewDelegate <NSObject>
- (void)tappedToFocusAtPoint:(CGPoint)point;
- (void)tappedToExposeAtPoint:(CGPoint)point;
- (void)tappedToResetFocusAndExposure;
@end

@interface THPreviewView : UIView

@property (strong, nonatomic) AVCaptureSession *session;
@property (weak, nonatomic) id<THPreviewViewDelegate> delegate;

@property (nonatomic) BOOL tapToFocusEnabled;
@property (nonatomic) BOOL tapToExposeEnabled;

@end
```

```
#import "THPreviewView.h"

@implementation THPreviewView

+ (Class)layerClass { // 1
    return [AVCaptureVideoPreviewLayer class];
}

- (void)setSession:(AVCaptureSession *)session { // 2
    [(AVCaptureVideoPreviewLayer*)self.layer setSession:session];
}

- (AVCaptureSession*)session {
    return [(AVCaptureVideoPreviewLayer*)self.layer session];
}

- (CGPoint)captureDevicePointForPoint:(CGPoint)point { // 3
    AVCaptureVideoPreviewLayer *layer =
        (AVCaptureVideoPreviewLayer *)self.layer;
    return [layer captureDevicePointOfInterestForPoint:point];
}

@end
```

```
#import <AVFoundation/AVFoundation.h>

extern NSString *const THThumbnailCreatedNotification;

@protocol THCameraControllerDelegate <NSObject> // 1
- (void)deviceConfigurationFailedWithError:(NSError *)error;
- (void)mediaCaptureFailedWithError:(NSError *)error;
- (void)assetLibraryWriteFailedWithError:(NSError *)error;
@end

@interface THCameraController : NSObject

@property (weak, nonatomic) id<THCameraControllerDelegate> delegate;
@property (nonatomic, strong, readonly) AVCaptureSession *captureSession;

// Session Configuration // 2
- (BOOL)setupSession:(NSError **)error;
- (void)startSession;
- (void)stopSession;

// Camera Device Support // 3
- (BOOL)switchCameras;
- (BOOL)canSwitchCameras;
@property (nonatomic, readonly) NSUInteger cameraCount;
@property (nonatomic, readonly) BOOL cameraHasTorch;
@property (nonatomic, readonly) BOOL cameraHasFlash;
@property (nonatomic, readonly) BOOL cameraSupportsTapToFocus;
@property (nonatomic, readonly) BOOL cameraSupportsTapToExpose;
@property (nonatomic) AVCaptureTorchMode torchMode;
@property (nonatomic) AVCaptureFlashMode flashMode;

// Tap to * Methods // 4
- (void)focusAtPoint:(CGPoint)point;
- (void)exposeAtPoint:(CGPoint)point;
- (void)resetFocusAndExposureModes;

/** Media Capture Methods **/ // 5

// Still Image Capture
- (void)captureStillImage;

// Video Recording
- (void)startRecording;
- (void)stopRecording;
- (BOOL)isRecording;

@end
```

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>
#import <AssetsLibrary/AssetsLibrary.h>
#import "NSFileManager+THAdditions.h"

@interface THCameraController : NSObject

@property (strong, nonatomic) dispatch_queue_t videoQueue;
@property (strong, nonatomic) AVCaptureSession *captureSession;
@property (weak, nonatomic) AVCaptureDeviceInput *activeVideoInput;

@property (strong, nonatomic) AVCaptureStillImageOutput *imageOutput;
@property (strong, nonatomic) AVCaptureMovieFileOutput *movieOutput;
@property (strong, nonatomic) NSURL *outputURL;

@end

@implementation THCameraController

- (BOOL)setupSession:(NSError ***)error {

    self.captureSession = [[AVCaptureSession alloc] init]; // 1
    self.captureSession.sessionPreset = AVCaptureSessionPresetHigh;
```

```
// Set up default camera device
AVCaptureDevice *videoDevice =
    [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo]; // 2

AVCaptureDeviceInput *videoInput = // 3
    [AVCaptureDeviceInput deviceInputWithDevice:videoDevice error:&error];
if (videoInput) {
    if ([self.captureSession canAddInput:videoInput]) { // 4
        [self.captureSession addInput:videoInput];
        self.activeVideoInput = videoInput;
    }
} else {
    return NO;
}

// Setup default microphone
AVCaptureDevice *audioDevice =
    [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeAudio]; // 5

AVCaptureDeviceInput *audioInput = // 6
    [AVCaptureDeviceInput deviceInputWithDevice:audioDevice error:&error];
if (audioInput) {
    if ([self.captureSession canAddInput:audioInput]) { // 7
        [self.captureSession addInput:audioInput];
    }
} else {
    return NO;
}
```

```
// Set up the still image output
self.imageOutput = [[AVCaptureStillImageOutput alloc] init]; // 8
self.imageOutput.outputSettings = @{@"AVVideoCodecKey" : AVVideoCodecJPEG};

if ([self.captureSession canAddOutput:self.imageOutput]) {
    [self.captureSession addOutput:self.imageOutput];
}

// Set up movie file output
self.movieOutput = [[AVCaptureMovieFileOutput alloc] init]; // 9

if ([self.captureSession canAddOutput:self.movieOutput]) {
    [self.captureSession addOutput:self.movieOutput];
}

self.videoQueue = dispatch_queue_create("com.tapharmonic.VideoQueue", NULL);

return YES;

}
```

```
- (void)startSession {
    if (![self.captureSession isRunning]) { // 1
        dispatch_async(self.videoQueue, ^{
            [self.captureSession startRunning];
        });
    }
}

- (void)stopSession {
    if ([self.captureSession isRunning]) { // 2
        dispatch_async(self.videoQueue, ^{
            [self.captureSession stopRunning];
        });
    }
}
```

NSLocalizedFailureReason=This app is not authorized to use iPhone Microphone.

```
- (AVCaptureDevice *)cameraWithPosition:(AVCaptureDevicePosition)position { // 1
    NSArray *devices = [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo];
    for (AVCaptureDevice *device in devices) {
        if (device.position == position) {
            return device;
        }
    }
    return nil;
}

- (AVCaptureDevice *)activeCamera { // 2
    return self.activeVideoInput.device;
}

- (AVCaptureDevice *)inactiveCamera { // 3
    AVCaptureDevice *device = nil;
    if (self.cameraCount > 1) {
        if ([self activeCamera].position == AVCaptureDevicePositionBack) {
            device = [self cameraWithPosition:AVCaptureDevicePositionFront];
        } else {
            device = [self cameraWithPosition:AVCaptureDevicePositionBack];
        }
    }
    return device;
}

- (BOOL)canSwitchCameras { // 4
    return self.cameraCount > 1;
}

- (NSUInteger)cameraCount { // 5
    return [[AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo] count];
}
```

```
- (BOOL)switchCameras {

    if (![self canSwitchCameras]) { // 1
        return NO;
    }

    NSError *error;
    AVCaptureDevice *videoDevice = [self inactiveCamera]; // 2

    AVCaptureDeviceInput *videoInput =
        [AVCaptureDeviceInput deviceInputWithDevice:videoDevice error:&error];

    if (videoInput) {
        [self.captureSession beginConfiguration]; // 3

        [self.captureSession removeInput:self.activeVideoInput]; // 4

        if ([self.captureSession canAddInput:videoInput]) { // 5
            [self.captureSession addInput:videoInput];
            self.activeVideoInput = videoInput;
        } else {
            [self.captureSession addInput:self.activeVideoInput];
        }

        [self.captureSession commitConfiguration]; // 6
    } else {
        [self.delegate deviceConfigurationFailedWithError:error]; // 7
        return NO;
    }

    return YES;
}
```

```
AVCaptureDevice *device = // Active video capture device  
  
if ([device isFocusModeSupported:AVCaptureFocusModeAutoFocus]) {  
    // Perform configuration  
}
```

```
AVCaptureDevice *device = // Active video capture device
if ([device isFocusModeSupported:AVCaptureFocusModeAutoFocus]) {
    NSError *error;
    if ([device lockForConfiguration:&error]) {
        device.focusMode = AVCaptureFocusModeAutoFocus;
        [device unlockForConfiguration];
    } else {
        // handle error
    }
}
```

```
- (BOOL)cameraSupportsTapToFocus { // 1
    return [[self activeCamera] isFocusPointOfInterestSupported];
}

- (void)focusAtPoint:(CGPoint)point { // 2
    AVCaptureDevice *device = [self activeCamera];

    if (device.isFocusPointOfInterestSupported && // 3
        [device isFocusModeSupported:AVCaptureFocusModeAutoFocus]) {

        NSError *error;
        if ([device lockForConfiguration:&error]) { // 4
            device.focusPointOfInterest = point;
            device.focusMode = AVCaptureFocusModeAutoFocus;
            [device unlockForConfiguration];
        } else {
            [self.delegate deviceConfigurationFailedWithError:error];
        }
    }
}
```

```
- (BOOL)cameraSupportsTapToExpose { // 1
    return [[self activeCamera] isExposurePointOfInterestSupported];
}

// Define KVO context pointer for observing 'adjustingExposure' device property.
static const NSString *THCameraAdjustingExposureContext;

- (void)exposeAtPoint:(CGPoint)point {
    AVCaptureDevice *device = [self activeCamera];

    AVCaptureExposureMode exposureMode =
        AVCaptureExposureModeContinuousAutoExposure;

    if (device.isExposurePointOfInterestSupported && // 2
        [device isExposureModeSupported:exposureMode]) {

        NSError *error;
        if ([device lockForConfiguration:&error]) { // 3

            device.exposurePointOfInterest = point;
            device.exposureMode = exposureMode;

            if ([device isExposureModeSupported:AVCaptureExposureModeLocked]) {
                [device addObserver:self // 4
                    forKeyPath:@"adjustingExposure"
                    options:NSMutableArrayObservingOptionNew
                    context:&THCameraAdjustingExposureContext];
            }
        }
    }
}
```

```

        }

        [device unlockForConfiguration];
    } else {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
}

- (void)observeValueForKeyPath:(NSString *)keyPath
    withObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {

    if (context == &THCameraAdjustingExposureContext) { // 5

        AVCaptureDevice *device = (AVCaptureDevice *)object;

        if (!device.isAdjustingExposure && // 6
            [device isExposureModeSupported:AVCaptureExposureModeLocked]) {

            [object removeObserver:self // 7
                forKeyPath:@"adjustingExposure"
                context:&THCameraAdjustingExposureContext];

            dispatch_async(dispatch_get_main_queue(), ^{
                NSError *error;
                if ([device lockForConfiguration:&error]) {
                    device.exposureMode = AVCaptureExposureModeLocked;
                    [device unlockForConfiguration];
                } else {
                    [self.delegate deviceConfigurationFailedWithError:error];
                }
            });
        }
    } else {
        [super observeValueForKeyPath:keyPath
            withObject:object
            change:change
            context:context];
    }
}

```

```
- (void)resetFocusAndExposureModes {
    AVCaptureDevice *device = [self activeCamera];

    AVCaptureFocusMode focusMode = AVCaptureFocusModeContinuousAutoFocus;

    BOOL canResetFocus = [device isFocusPointOfInterestSupported] && // 1
                         [device isFocusModeSupported:focusMode];

    AVCaptureExposureMode exposureMode =
        AVCaptureExposureModeContinuousAutoExposure;

    BOOL canResetExposure = [device isExposurePointOfInterestSupported] && // 2
                           [device isExposureModeSupported:exposureMode];

    CGPoint centerPoint = CGPointMake(0.5f, 0.5f); // 3

    NSError *error;
    if ([device lockForConfiguration:&error]) {

        if (canResetFocus) { // 4
            device.focusMode = focusMode;
            device.focusPointOfInterest = centerPoint;
        }

        if (canResetExposure) { // 5
            device.exposureMode = exposureMode;
            device.exposurePointOfInterest = centerPoint;
        }

        [device unlockForConfiguration];
    } else {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
}
```

```
- (BOOL)cameraHasFlash {
    return [[self activeCamera] hasFlash];
}

- (AVCaptureFlashMode)flashMode {
    return [[self activeCamera] flashMode];
}

- (void)setFlashMode:(AVCaptureFlashMode)flashMode {
    AVCaptureDevice *device = [self activeCamera];

    if ([[device isFlashModeSupported:flashMode]]) {

        NSError *error;
        if ([device lockForConfiguration:&error]) {
            device.flashMode = flashMode;
            [device unlockForConfiguration];
        } else {
            [self.delegate deviceConfigurationFailedWithError:error];
        }
    }
}

- (BOOL)cameraHasTorch {
    return [[self activeCamera] hasTorch];
}
```

```
- (AVCaptureTorchMode)torchMode {
    return [[self activeCamera] torchMode];
}

- (void)setTorchMode:(AVCaptureTorchMode)torchMode {
    AVCaptureDevice *device = [self activeCamera];

    if ([[device isTorchModeSupported:torchMode]]) {

        NSError *error;
        if ([[device lockForConfiguration:&error]]) {
            device.torchMode = torchMode;
            [device unlockForConfiguration];
        } else {
            [self.delegate deviceConfigurationFailedWithError:error];
        }
    }
}
```

```
AVCaptureConnection *connection = // Active video capture connection
id completionHandler = ^(CMSampleBufferRef buffer, NSError *error){
    // Handle image capture
};

[imageOutput captureStillImageAsynchronouslyFromConnection:connection
completionHandler:completionHandler];
```

```
- (void)captureStillImage {
    AVCaptureConnection *connection = // 1
        [self.imageOutput connectionWithMediaType:AVMediaTypeVideo];

    if (connection.isVideoOrientationSupported) { // 2
        connection.videoOrientation = [self currentVideoOrientation];
    }

    id handler = ^(CMSampleBufferRef sampleBuffer, NSError *error) {
        if (sampleBuffer != NULL) {

            NSData *imageData = // 4
                [AVCaptureStillImageOutput
                 jpegStillImageNSDataRepresentation:sampleBuffer];

            UIImage *image = [[UIImage alloc] initWithData:imageData]; // 5

        } else {
            NSLog(@"NULL sampleBuffer: %@", [error localizedDescription]);
        }
    };
    // Capture still image // 6
    [self.imageOutput captureStillImageAsynchronouslyFromConnection:connection
        completionHandler:handler];
}
```

```
- (AVCaptureVideoOrientation)currentVideoOrientation {

    AVCaptureVideoOrientation orientation;

    switch ([UIDevice currentDevice].orientation) { // 3
        case UIDeviceOrientationPortrait:
            orientation = AVCaptureVideoOrientationPortrait;
            break;
        case UIDeviceOrientationLandscapeRight:
            orientation = AVCaptureVideoOrientationLandscapeLeft;
            break;
        case UIDeviceOrientationPortraitUpsideDown:
            orientation = AVCaptureVideoOrientationPortraitUpsideDown;
            break;
        default:
            orientation = AVCaptureVideoOrientationLandscapeRight;
            break;
    }

    return orientation;
}
```

```
ALAuthorizationStatus status = [ALAssetsLibrary authorizationStatus];
if (status == ALAuthorizationStatusDenied) {
    // Show prompt indicating the application won't function
    // correctly without access to the library
} else {
    // Perform authorized access to the library
}
```

```
- (void)captureStillImage {
    AVCaptureConnection *connection =
        [self.imageOutput connectionWithMediaType:AVMediaTypeVideo];

    if (connection.isVideoOrientationSupported) {
        connection.videoOrientation = [self currentVideoOrientation];
    }

    id handler = ^(CMSampleBufferRef sampleBuffer, NSError *error) {
        if (sampleBuffer != NULL) {

            NSData *imageData =
                [AVCaptureStillImageOutput
                    jpegStillImageNSDataRepresentation:sampleBuffer];

            UIImage *image = [[UIImage alloc] initWithData:imageData];
            [self writeImageToAssetsLibrary:image]; // 1

        } else {
            NSLog(@"%@", @"NULL sampleBuffer: %@", [error localizedDescription]);
        }
    };
    // Capture still image
    [self.imageOutput captureStillImageAsynchronouslyFromConnection:connection
        completionHandler:handler];
}
```

```
- (void)writeImageToAssetsLibrary:(UIImage *)image {
    ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init]; // 2
    [library writeImageToSavedPhotosAlbum:image.CGImage // 3
        orientation:(NSInteger)image.imageOrientation // 4
        completionBlock:^(NSURL *assetURL, NSError *error) {
            if (!error) {
                [self postThumbnailNotification:image]; // 5
            } else {
                id message = [error localizedDescription];
                NSLog(@"Error: %@", message);
            }
        }];
}

- (void)postThumbnailNotification:(UIImage *)image {
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc postNotificationName:THThumbnailCreatedNotification object:image];
}
```

```
- (BOOL)isRecording { // 1
    return self.movieOutput.isRecording;
}

- (void)startRecording {
    if (![self isRecording]) {

        AVCaptureConnection *videoConnection = // 2
            [self.movieOutput connectionWithMediaType:AVMediaTypeVideo];

        if ([videoConnection isVideoOrientationSupported]) { // 3
            videoConnection.videoOrientation = self.currentVideoOrientation;
        }

        if ([videoConnection isVideoStabilizationSupported]) { // 4
            videoConnection.enablesVideoStabilizationWhenAvailable = YES;
        }
    }

    AVCaptureDevice *device = [self activeCamera];

    if (device.isSmoothAutoFocusSupported) { // 5
        NSError *error;
        if ([device lockForConfiguration:&error]) {
            device.smoothAutoFocusEnabled = YES;
            [device unlockForConfiguration];
        } else {
            [self.delegate deviceConfigurationFailedWithError:error];
        }
    }
}
```

```
        }
    }

    self.outputURL = [self uniqueURL]; // 6
    [self.movieOutput startRecordingToOutputFileURL:self.outputURL // 8
                                                recordingDelegate:self];

}

- (NSURL *)uniqueURL { // 7

    NSFileManager *fileManager = [NSFileManager defaultManager];
    NSString *dirPath =
        [fileManager temporaryDirectoryWithTemplateString:@"kamera.XXXXXX"];

    if (dirPath) {
        NSString *filePath =
            [dirPath stringByAppendingPathComponent:@"kamera_movie.mov"];
        return [NSURL fileURLWithPath:filePath];
    }

    return nil;
}

- (void)stopRecording { // 9
    if ([self isRecording]) {
        [self.movieOutput stopRecording];
    }
}
```

```
@interface THCameraController () <AVCaptureFileOutputRecordingDelegate>

@property (strong, nonatomic) AVCaptureSession *captureSession;
@property (weak, nonatomic) AVCaptureDeviceInput *activeVideoInput;

@property (strong, nonatomic) AVCaptureStillImageOutput *imageOutput;
@property (strong, nonatomic) AVCaptureMovieFileOutput *movieOutput;
@property (strong, nonatomic) NSURL *outputURL;

@end
```

```
- (void)captureOutput:(AVCaptureFileOutput *)captureOutput
didFinishRecordingToOutputFileAtURL:(NSURL *)outputFileURL
    fromConnections:(NSArray *)connections
        error:(NSError *)error {
    if (error) { // 1
        [self.delegate mediaCaptureFailedWithError:error];
    } else {
        [self writeVideoToAssetsLibrary:[self.outputURL copy]];
    }
    self.outputURL = nil;
}

- (void)writeVideoToAssetsLibrary:(NSURL *)videoURL {

    ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init]; // 2

    if ([library videoAtPathIsCompatibleWithSavedPhotosAlbum:videoURL]) { // 3
        ALAssetsLibraryWriteVideoCompletionBlock completionBlock;

        completionBlock = ^(NSURL *assetURL, NSError *error){ // 4
            if (error) {
                [self.delegate assetLibraryWriteFailedWithError:error];
            } else {
                [self generateThumbnailForVideoAtURL:videoURL];
            }
        };
    }
}
```

```
[library writeVideoAtPathToSavedPhotosAlbum:videoURL           // 8
                                         completionBlock:completionBlock];
}

}

- (void)generateThumbnailForVideoAtURL:(NSURL *)videoURL {
    dispatch_async(self.videoQueue, ^{
        AVAsset *asset = [AVAsset assetWithURL:videoURL];

        AVAssetImageGenerator *imageGenerator =                         // 5
            [AVAssetImageGenerator assetImageGeneratorWithAsset:asset];
        imageGenerator.maximumSize = CGSizeMake(100.0f, 0.0f);
        imageGenerator.appliesPreferredTrackTransform = YES;

        CGImageRef imageRef = [imageGenerator copyCGImageAtTime:kCMTimeZero // 6
                                         actualTime:NULL
                                         error:nil];
        UIImage *image = [UIImage imageWithCGImage:imageRef];
        CGImageRelease(imageRef);

        dispatch_async(dispatch_get_main_queue(), ^{
            // 7
            [self postThumbnailNotification:image];
        });
    });
}
```

```
#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THCameraZoomingDelegate <NSObject> // 1
- (void)rampedZoomToValue:(CGFloat)value;
@end

@interface THCameraController : THBaseCameraController

@property (weak, nonatomic) id<THCameraZoomingDelegate> zoomingDelegate;

- (BOOL)cameraSupportsZoom; // 2

- (void)setZoomValue:(CGFloat)zoomValue;
- (void)rampZoomToValue:(CGFloat)zoomValue;
- (void)cancelZoom;

@end
```

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

const CGFloat THZoomRate = 1.0f;

// KVO Contexts
static const NSString *THRampingVideoZoomContext;
static const NSString *THRampingVideoZoomFactorContext;

@implementation THCameraController

- (BOOL)cameraSupportsZoom {
    return self.activeCamera.activeFormat.videoMaxZoomFactor > 1.0f;           // 1
}

- (CGFloat)maxZoomFactor {
    return MIN(self.activeCamera.activeFormat.videoMaxZoomFactor, 4.0f);        // 2
}

- (void)setZoomValue:(CGFloat)zoomValue {                                         // 3
    if (!self.activeCamera.isRampingVideoZoom) {

        NSError *error;
        if ([self.activeCamera lockForConfiguration:&error]) {                  // 4

            // Provide linear feel to zoom slider
            CGFloat zoomFactor = pow([self maxZoomFactor], zoomValue);          // 5
            self.activeCamera.videoZoomFactor = zoomFactor;

            [self.activeCamera unlockForConfiguration];                          // 6
        } else {
            [self.delegate deviceConfigurationFailedWithError:error];
        }
    }
}

...
@end
```

```
- (void)rampZoomToValue:(CGFloat)zoomValue { // 1
    CGFloat zoomFactor = pow([self maxZoomFactor], zoomValue);
    NSError *error;
    if ([self.activeCamera lockForConfiguration:&error]) {
        [self.activeCamera rampToVideoZoomFactor:zoomFactor
                                         withRate:THZoomRate];
        [self.activeCamera unlockForConfiguration];
    } else {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
}

- (void)cancelZoom { // 3
    NSError *error;
    if ([self.activeCamera lockForConfiguration:&error]) {
        [self.activeCamera cancelVideoZoomRamp];
        [self.activeCamera unlockForConfiguration];
    } else {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
}
```

```
- (BOOL)setupSessionInputs:(NSError **)error {
    BOOL success = [super setupSessionInputs:error]; // 1
    if (success) {
        [self.activeCamera addObserver:self
            forKeyPath:@"videoZoomFactor"
            options:0
            context:&THRampingVideoZoomFactorContext];
        [self.activeCamera addObserver:self
            forKeyPath:@"rampingVideoZoom"
            options:0
            context:&THRampingVideoZoomContext];
    }
    return success;
}

- (void)observeValueForKeyPath:(NSString *)keyPath
    withObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context {
    if (context == &THRampingVideoZoomContext) {
        [self updateZoomingDelegate]; // 2
    } else if (context == &THRampingVideoZoomFactorContext) {
        if (self.activeCamera.isRampingVideoZoom) {
            [self updateZoomingDelegate]; // 3
        }
    } else {
        [super observeValueForKeyPath:keyPath
            withObject:object
            change:change
            context:context];
    }
}

- (void)updateZoomingDelegate {
    CGFloat curZoomFactor = self.activeCamera.videoZoomFactor;
    CGFloat maxZoomFactor = [self maxZoomFactor];
    CGFloat value = log(curZoomFactor) / log(maxZoomFactor); // 4
    [self.zoomingDelegate rampedZoomToValue:value]; // 5
}
```

```
#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THFaceDetectionDelegate <NSObject>
- (void)didDetectFaces:(NSArray *)faces;
@end

@interface THCameraController : THBaseCameraController

@property (weak, nonatomic) id <THFaceDetectionDelegate> faceDetectionDelegate;

@end
```

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController ()
@property (strong, nonatomic) AVCaptureMetadataOutput *metadataOutput;      // 1
@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError **)error {

    self.metadataOutput = [[AVCaptureMetadataOutput alloc] init];           // 2

    if ([self.captureSession canAddOutput:self.metadataOutput]) {
        [self.captureSession addOutput:self.metadataOutput];

        NSArray *metadataObjectTypes = @+[AVMetadataObjectTypeFace];          // 3
        self.metadataOutput.metadataObjectTypes = metadataObjectTypes;

        dispatch_queue_t mainQueue = dispatch_get_main_queue();
        [self.metadataOutput setMetadataObjectsDelegate:self
                                         queue:mainQueue];           // 4

    }

    return YES;
}

} else {                                              // 5
    if (error) {
        NSDictionary *userInfo = @{@"NSLocalizedDescriptionKey":
                                     @"Failed to still image output."};
        *error = [NSError errorWithDomain:THCameraErrorDomain
                                    code:THCameraErrorFailedToAddOutput
                                    userInfo:userInfo];
    }
    return NO;
}
}

@end
```

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController ()<AVCaptureMetadataOutputObjectsDelegate>           // 1
...
@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError **)error {
...
}

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputMetadataObjects:(NSArray *)metadataObjects
fromConnection:(AVCaptureConnection *)connection {

    for (AVMetadataFaceObject *face in metadataObjects) {                                // 2
        NSLog(@"Face detected with ID: %li", (long)face.faceID);
        NSLog(@"Face bounds: %@", NSStringFromCGRect(face.bounds));
    }
}

[self.faceDetectionDelegate didDetectFaces:metadataObjects];                         // 3
}

@end
```

```
#import <AVFoundation/AVFoundation.h>
#import "THFaceDetectionDelegate.h"

@interface THPreviewView : UIView <THFaceDetectionDelegate>

@property (strong, nonatomic) AVCaptureSession *session;

@end
```

```
#import "THPreviewView.h"

@interface THPreviewView () // 1
@property (strong, nonatomic) CALayer *overlayLayer;
@property (strong, nonatomic) NSMutableDictionary *faceLayers;
@property (nonatomic, readonly) AVCaptureVideoPreviewLayer *previewLayer;
@end

@implementation THPreviewView

+ (Class)layerClass { // 2
    return [AVCaptureVideoPreviewLayer class];
}

- (id)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        [self setupView];
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder];
    if (self) {
        [self setupView];
    }
    return self;
}
```

```
}

- (void)setupView {
    //
}

- (AVCaptureSession*)session {
    return self.previewLayer.session;
}

- (void)setSession:(AVCaptureSession *)session { // 3
    self.previewLayer.session = session;
}

- (AVCaptureVideoPreviewLayer *)previewLayer {
    return (AVCaptureVideoPreviewLayer *)self.layer;
}

- (void)didDetectFaces:(NSArray *)faces {
    //
}

@end
```

```
@implementation THPreviewView

...
- (void)setupView {
    self.faceLayers = [NSMutableDictionary dictionary]; // 1
    self.previewLayer.videoGravity = AVLayerVideoGravityResizeAspectFill;

    self.overlayLayer = [CALayer layer]; // 2
    self.overlayLayer.frame = self.bounds;
    self.overlayLayer.sublayerTransform = THMakePerspectiveTransform(1000);
    [self.previewLayer addSublayer:self.overlayLayer];
}

static CATransform3D THMakePerspectiveTransform(CGFloat eyePosition) { // 3
    CATransform3D transform = CATransform3DIdentity;
    transform.m34 = -1.0 / eyePosition;
    return transform;
}

...
@end
```

```
@implementation THPreviewView

...
- (void)detectFaces:(NSArray *)faces {
    NSArray *transformedFaces = [self transformedFacesFromFaces:faces]; // 1
    // Process transformed faces. To be implemented in Listing 7.12.
}

- (NSArray *)transformedFacesFromFaces:(NSArray *)faces { // 2
    NSMutableArray *transformedFaces = [NSMutableArray array];
    for (AVMetadataObject *face in faces) {
        AVMetadataObject *transformedFace = // 3
            [self.previewLayer transformedMetadataObjectForMetadataObject:face];
        [transformedFaces addObject:transformedFace];
    }
    return transformedFaces;
}

...
@end
```

```
@implementation THPreviewView

...
- (void)detectFaces:(NSArray *)faces {
    NSArray *transformedFaces = [self transformedFacesFromFaces:faces];
    NSMutableArray *lostFaces = [self.faceLayers.allKeys mutableCopy]; // 1

    for (AVMetadataFaceObject *face in transformedFaces) {
        NSNumber *faceID = @(face.faceID); // 2
        [lostFaces removeObject:faceID];

        CALayer *layer = self.faceLayers[faceID]; // 3
        if (!layer) {
            // no layer for faceID, create new face layer
            layer = [self makeFaceLayer]; // 4
            [self.overlayLayer addSublayer:layer];
            self.faceLayers[faceID] = layer;
        }

        layer.transform = CATransform3DIdentity; // 6
        layer.frame = face.bounds;
    }
}
```

```
    for (NSNumber *faceID in lostFaces) {                                // 7
        CALayer *layer = self.faceLayers[faceID];
        [layer removeFromSuperlayer];
        [self.faceLayers removeObjectForKey:faceID];
    }

}

- (CALayer *)makeFaceLayer {                                         // 5
    CALayer *layer = [CALayer layer];
    layer.borderWidth = 5.0f;
    layer.borderColor =
        [UIColor colorWithRed:0.188 green:0.517 blue:0.877 alpha:1.000].CGColor;
    return layer;
}

...
@end
```

```
@implementation THPreviewView
...
- (void)detectFaces:(NSArray *)faces {
    NSArray *transformedFaces = [self transformedFacesFromFaces:faces];
    NSMutableArray *lostFaces = [self.faceLayers.allKeys mutableCopy];
    for (AVMetadataFaceObject *face in transformedFaces) {
        NSNumber *faceID = @(face.faceID);
        [lostFaces removeObject:faceID];
        CALayer *layer = self.faceLayers[faceID];
        if (!layer) {
            // no layer for faceID, create new face layer
            layer = [self makeFaceLayer];
            [self.overlayLayer addSublayer:layer];
            self.faceLayers[faceID] = layer;
        }
        layer.transform = CATransform3DIdentity; // 1
        layer.frame = face.bounds;
        if (face.hasRollAngle) {
            CATransform3D t = [self transformForRollAngle:face.rollAngle]; // 2
            layer.transform = CATransform3DConcat(layer.transform, t);
        }
        if (face.hasYawAngle) {
            CATransform3D t = [self transformForYawAngle:face.yawAngle]; // 4
            layer.transform = CATransform3DConcat(layer.transform, t);
        }
    }
    for (NSNumber *faceID in lostFaces) { // 6
        CALayer *layer = self.faceLayers[faceID];
        [layer removeFromSuperlayer];
        [self.faceLayers removeObjectForKey:faceID];
    }
}
```

```

// Rotate around Z-axis
- (CATransform3D)transformForRollAngle:(CGFloat)rollAngleInDegrees {           // 3
    CGFloat rollAngleInRadians = THDegreesToRadians(rollAngleInDegrees);
    return CATransform3DMakeRotation(rollAngleInRadians, 0.0f, 0.0f, 1.0f);
}

// Rotate around Y-axis
- (CATransform3D)transformForYawAngle:(CGFloat)yawAngleInDegrees {           // 5
    CGFloat yawAngleInRadians = THDegreesToRadians(yawAngleInDegrees);

    CATransform3D yawTransform =
        CATransform3DMakeRotation(yawAngleInRadians, 0.0f, -1.0f, 0.0f);

    return CATransform3DConcat(yawTransform, [self orientationTransform]);
}

- (CATransform3D)orientationTransform {                                       // 6
    CGFloat angle = 0.0;
    switch ([UIDevice currentDevice].orientation) {
        case UIDeviceOrientationPortraitUpsideDown:
            angle = M_PI;
            break;
        case UIDeviceOrientationLandscapeRight:
            angle = -M_PI / 2.0f;
            break;
        case UIDeviceOrientationLandscapeLeft:
            angle = M_PI / 2.0f;
            break;
        default: // as UIDeviceOrientationPortrait
            angle = 0.0;
            break;
    }
    return CATransform3DMakeRotation(angle, 0.0f, 0.0f, 1.0f);
}

static CGFloat THDegreesToRadians(CGFloat degrees) {
    return degrees * M_PI / 180;
}

...
@end

```

```
#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THCodeDetectionDelegate <NSObject>
- (void)didDetectCodes:(NSArray *)codes;
@end

@interface THCameraController : THBaseCameraController

@property (weak, nonatomic) id <THCodeDetectionDelegate> codeDetectionDelegate;

@end
```

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>
@interface THCameraController () <AVCaptureMetadataOutputObjectsDelegate> // 1
@property (strong, nonatomic) AVCaptureMetadataOutput *metadataOutput;
@end

@implementation THCameraController

- (NSString *)sessionPreset { // 2
    return AVCaptureSessionPreset640x480;
}

- (BOOL)setupSessionInputs:(NSError *__autoreleasing *)error {
    BOOL success = [super setupSessionInputs:error];
    if (success) {
        if ([self.activeCamera autoFocusRangeRestrictionSupported]) { // 3

            if ([self.activeCamera lockForConfiguration:error]) {

                self.activeCamera.autoFocusRangeRestriction =
                    AVCaptureAutoFocusRangeRestrictionNear;

                [self.activeCamera unlockForConfiguration];
            }
        }
    }
    return success;
}

...
@end
```

```
@implementation THCameraController

...
- (BOOL)setupSessionOutputs:(NSError **)error {
    self.metadataOutput = [[AVCaptureMetadataOutput alloc] init];
    if ([self.captureSession canAddOutput:self.metadataOutput]) {
        [self.captureSession addOutput:self.metadataOutput];
        dispatch_queue_t mainQueue = dispatch_get_main_queue();
        [self.metadataOutput setMetadataObjectsDelegate:self
                                         queue:mainQueue];
        NSArray *types = @+[AVMetadataObjectTypeQRCode, // 1
                           AVMetadataObjectTypeAztecCode];
        self.metadataOutput.metadataObjectTypes = types;
    } else {
        NSDictionary *userInfo = @{@"NSLocalizedDescriptionKey":
                                   @"Failed to add metadata output."};
        *error = [NSError errorWithDomain:THCameraErrorDomain
                                     code:THCameraErrorFailedToAddOutput
                                     userInfo:userInfo];
        return NO;
    }
    return YES;
}

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputMetadataObjects:(NSArray *)metadataObjects
fromConnection:(AVCaptureConnection *)connection {
    [self.codeDetectionDelegate didDetectCodes:metadataObjects]; // 2
}
@end
```

```
#import <AVFoundation/AVFoundation.h>
#import "THCodeDetectionDelegate.h"

@interface THPreviewView : UIView <THCodeDetectionDelegate>

@property (strong, nonatomic) AVCaptureSession *session;

@end
```

```
#import "THPreviewView.h"

@interface THPreviewView () // 1
@property (strong, nonatomic) NSMutableDictionary *codeLayers;
@end

@implementation THPreviewView

+ (Class)layerClass { // 2
    return [AVCaptureVideoPreviewLayer class];
}

- (id)initWithFrame:(CGRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        [self setupView];
    }
    return self;
}

- (id)initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder];
    if (self) {
        [self setupView];
    }
}
```

```
        }
        return self;
    }

- (void)setupView {                                // 3
    _codeLayers = [NSMutableDictionary dictionary];
    self.previewLayer.videoGravity = AVLayerVideoGravityResizeAspect;
}

- (AVCaptureSession*)session {
    return self.previewLayer.session;
}

- (void)setSession:(AVCaptureSession *)session {      // 4
    self.previewLayer.session = session;
}

- (AVCaptureVideoPreviewLayer *)previewLayer {
    return (AVCaptureVideoPreviewLayer *)self.layer;
}

- (void)didDetectCodes:(NSArray *)codes {

}

@end
```

```
@implementation THPreviewView

...
- (void)didDetectCodes:(NSArray *)codes {
    NSArray *transformedCodes = [self transformedCodesFromCodes:codes]; // 1
}

- (NSArray *)transformedCodesFromCodes:(NSArray *)codes { // 2
    NSMutableArray *transformedCodes = [NSMutableArray array];
    for (AVMetadataObject *code in codes) {
        AVMetadataObject *transformedCode =
            [self.previewLayer transformedMetadataObjectForMetadataObject:code];
        [transformedCodes addObject:transformedCode];
    }
    return transformedCodes;
}

@end
```

```
@implementation THPreviewView

...
- (void)didDetectCodes:(NSArray *)codes {
    NSArray *transformedCodes = [self transformedCodesFromCodes:codes];
    NSMutableArray *lostCodes = [self.codeLayers.allKeys mutableCopy]; // 1
    for (AVMetadataMachineReadableCodeObject *code in transformedCodes) {
        NSString *stringValue = code.stringValue; // 2
        if (stringValue) {
            [lostCodes removeObject:stringValue];
        } else {
            continue;
        }
        NSArray *layers = self.codeLayers[stringValue]; // 3
        if (!layers) {
            // no layers for stringValue, create new code layers
            layers = @[[self makeBoundsLayer], [self makeCornersLayer]];
            self.codeLayers[stringValue] = layers;
            [self.previewLayer addSublayer:layers[0]];
            [self.previewLayer addSublayer:layers[1]];
        }
    }
}
```

```

        CAShapeLayer *boundsLayer = layers[0];                                // 4
        boundsLayer.path  = [self bezierPathForBounds:code.bounds].CGPath;

        NSLog(@"%@", stringValue);                                         // 5
    }

    for (NSString *stringValue in lostCodes) {                                // 6
        for (CALayer *layer in self.codeLayers[stringValue]) {
            [layer removeFromSuperlayer];
        }
        [self.codeLayers removeObjectForKey:stringValue];
    }
}

- (UIBezierPath *)bezierPathForBounds:(CGRect)bounds {
    return [UIBezierPath bezierPathWithRect:bounds];
}

- (CAShapeLayer *)makeBoundsLayer {
    CAShapeLayer *shapeLayer = [CAShapeLayer layer];
    shapeLayer.strokeColor =
        [UIColor colorWithRed:0.95f green:0.75f blue:0.06f alpha:1.0f].CGColor;
    shapeLayer.fillColor = nil;
    shapeLayer.lineWidth = 4.0f;
    return shapeLayer;
}

- (CAShapeLayer *)makeCornersLayer {
    CAShapeLayer *cornersLayer = [CAShapeLayer layer];
    cornersLayer.lineWidth = 2.0f;
    cornersLayer.strokeColor =
        [UIColor colorWithRed:0.172 green:0.671 blue:0.428 alpha:1.000].CGColor;
    cornersLayer.fillColor =
        [UIColor colorWithRed:0.190 green:0.753 blue:0.489 alpha:0.500].CGColor;

    return cornersLayer;
}

@end

```

```
@implementation THPreviewView

...
- (void)didDetectCodes:(NSArray *)codes {
    NSArray *transformedCodes = [self transformedCodesFromCodes:codes];
    NSMutableArray *lostCodes = [self.codeLayers.allKeys mutableCopy];
    for (AVMetadataMachineReadableCodeObject *code in transformedCodes) {
        NSString *stringValue = code.stringValue;
        if (stringValue) {
            [lostCodes removeObject:stringValue];
        } else {
            continue;
        }
        NSArray *layers = self.codeLayers[stringValue];
        if (!layers) {
            // no layers for stringValue, create new code layers
            layers = @[[self makeBoundsLayer], [self makeCornersLayer]];
            self.codeLayers[stringValue] = layers;
            [self.previewLayer addSublayer:layers[0]];
            [self.previewLayer addSublayer:layers[1]];
        }
    }
}
```

```

        }

        CAShapeLayer *boundsLayer = layers[0];
        boundsLayer.path = [self bezierPathForBounds:code.bounds].CGPath;

        CAShapeLayer *cornersLayer = layers[1]; // 1
        cornersLayer.path = [self bezierPathForCorners:code.corners].CGPath;

        NSLog(@"%@", stringValue);
    }

    for (NSString *stringValue in lostCodes) {
        for (CALayer *layer in self.codeLayers[stringValue]) {
            [layer removeFromSuperlayer];
        }
        [self.codeLayers removeObjectForKey:stringValue];
    }
}

- (UIBezierPath *)bezierPathForCorners:(NSArray *)corners {
    UIBezierPath *path = [UIBezierPath bezierPath];
    for (int i = 0; i < corners.count; i++) {
        CGPoint point = [self pointForCorner:corners[i]]; // 2
        if (i == 0) { // 4
            [path moveToPoint:point];
        } else {
            [path addLineToPoint:point];
        }
    }
    [path closePath]; // 5
    return path;
}

- (CGPoint)pointForCorner:(NSDictionary *)corner { // 3
    CGPoint point;
    CGPointMakeWithDictionaryRepresentation((CFDictionaryRef)corner, &point);
    return point;
}

...

```

@end

```
#import <AVFoundation/AVFoundation.h>

@interface AVCaptureDevice (THAdditions)

- (BOOL)supportsHighFrameRateCapture;
- (BOOL)enableHighFrameRateCapture:(NSError **)error;
```

```
@end
```

```
#import "AVCaptureDevice+THAdditions.h"
#import "TSError.h"

@interface THQualityOfService : NSObject

@property(strong, nonatomic, readonly) AVCaptureDeviceFormat *format;
@property(strong, nonatomic, readonly) AVFrameRateRange *frameRateRange;
@property(nonatomic, readonly) BOOL isHighFrameRate;

+ (instancetype)qosWithFormat:(AVCaptureDeviceFormat *)format
                      frameRateRange:(AVFrameRateRange *)frameRateRange;

- (BOOL)isHighFrameRate;

@end

@implementation THQualityOfService

+ (instancetype)qosWithFormat:(AVCaptureDeviceFormat *)format
                      frameRateRange:(AVFrameRateRange *)frameRateRange {

    return [[self alloc] initWithFormat:format frameRateRange:frameRateRange];
}

- (instancetype)initWithFormat:(AVCaptureDeviceFormat *)format
                      frameRateRange:(AVFrameRateRange *)frameRateRange {
    self = [super init];
    if (self) {
        _format = format;
        _frameRateRange = frameRateRange;
    }
    return self;
}

- (BOOL)isHighFrameRate {
    return self.frameRateRange.maxFrameRate > 30.0f;
}

@end
```

```
@implementation AVCaptureDevice (THAdditions)

- (BOOL)supportsHighFrameRateCapture {
    if (! [self hasMediaType:AVMediaTypeVideo]) { // 1
        return NO;
    }
    return [self findHighestQualityOfService].isHighFrameRate; // 2
}

- (THQualityOfService *)findHighestQualityOfService {

    AVCaptureDeviceFormat *maxFormat = nil;
    AVFrameRateRange *maxFrameRateRange = nil;

    for (AVCaptureDeviceFormat *format in self.formats) {

        FourCharCode codecType = // 3
            CMVideoFormatDescriptionGetCodecType(format.formatDescription);

        if (codecType == kCVPixelFormatType_420YpCbCr8BiPlanarVideoRange) {

            NSArray *frameRateRanges = format.videoSupportedFrameRateRanges;

            for (AVFrameRateRange *range in frameRateRanges) { // 4
                if (range.maxFrameRate > maxFrameRateRange.maxFrameRate) {
                    maxFormat = format;
                    maxFrameRateRange = range;
                }
            }
        }
    }

    return [THQualityOfService qosWithFormat:maxFormat // 6
                           frameRateRange:maxFrameRateRange];
}

...
@end
```

```
- (BOOL)enableMaxFrameRateCapture:(NSError **)error {

    THQualityOfService *qos = [self findHighestQualityOfService];

    if (!qos.isHighFrameRate) { // 1
        if (error) {
            NSString *message = @"Device does not support high FPS capture";
            NSDictionary *userInfo = @{@"NSLocalizedDescriptionKey" : message};

            NSUInteger code = THCameraErrorHighFrameRateCaptureNotSupported;

            *error = [NSError errorWithDomain:THCameraErrorDomain
                                         code:code
                                         userInfo:userInfo];
        }
        return NO;
    }

    if ([self lockForConfiguration:error]) { // 2

        CMTIME minFrameDuration = qos.frameRateRange.minFrameDuration;

        self.activeFormat = qos.format; // 3
        self.activeVideoMinFrameDuration = minFrameDuration; // 4
        self.activeVideoMaxFrameDuration = minFrameDuration;

        [self unlockForConfiguration];
        return YES;
    }
    return NO;
}
```

```
#import "THCameraController.h"
#import "AVCaptureDevice+THAdditions.h"

@implementation THCameraController

- (BOOL)cameraSupportsHighFrameRateCapture {
    return [self.activeCamera supportsHighFrameRateCapture];
}

- (BOOL)enableHighFrameRateCapture {
    NSError *error;
    BOOL enabled = [self.activeCamera enableMaxFrameRateCapture:&error];
    if (!enabled) {
        [self.delegate deviceConfigurationFailedWithError:error];
    }
    return enabled;
}

@end
```

```
const int BYTES_PER_PIXEL = 4;

CMSSampleBufferRef sampleBuffer = // obtained sample buffer

CVPixelBufferRef pixelBuffer =
    CMSSampleBufferGetImageBuffer(sampleBuffer); // 1

CVPixelBufferLockBaseAddress( pixelBuffer, 0); // 2

size_t bufferWidth = CVPixelBufferGetWidth(pixelBuffer); // 3
size_t bufferHeight = CVPixelBufferGetHeight(pixelBuffer);

unsigned char *pixel =
    (unsigned char *)CVPixelBufferGetBaseAddress(pixelBuffer);
unsigned char grayPixel;

for (int row = 0; row < bufferHeight; row++) { // 4
    for(int column = 0; column < bufferWidth; column++) {
        grayPixel = (pixel[0] + pixel[1] + pixel[2]) / 3;
        pixel[0] = pixel[1] = pixel[2] = grayPixel;
        pixel += BYTES_PER_PIXEL;
    }
}

CVPixelBufferUnlockBaseAddress(pixelBuffer, 0); // 5

// Process grayscale video frame
```

```
CMFormatDescriptionRef formatDescription =
    CMSampleBufferGetFormatDescription(sampleBuffer);

CMMediaType mediaType = CMFormatDescriptionGetMediaType(formatDescription);

if (mediaType == kCMMediaType_Video) {
    CVPixelBufferRef pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer);
    // Process the frame of video
} else if (mediaType == kCMMediaType_Audio) {
    CMBlockBufferRef blockBuffer = CMSampleBufferGetDataBuffer(sampleBuffer);
    // Process audio samples
}
```

```
CFDictionaryRef exifAttachments =  
    (CFDictionaryRef)CMGetAttachment(sampleBuffer,  
                                    kCGImagePropertyExifDictionary,  
                                    NULL);
```

```
{  
    ApertureValue = "2.526068811667587";  
    BrightnessValue = "-0.4554591284958377";  
    ExposureMode = 0;  
    ExposureProgram = 2;  
    ExposureTime = "0.04166666666666666";  
    FNumber = "2.4";  
    Flash = 32;  
    FocalLenIn35mmFilm = 35;  
    FocalLength = "2.18";  
    ISOSpeedRatings = (  
        800  
    );  
    LensMake = Apple;  
    LensModel = "iPhone 5 front camera 2.18mm f/2.4";  
    LensSpecification = (  
        "2.18",  
        "2.18",  
        "2.4",  
        "2.4"  
    );  
    MeteringMode = 5;  
    PixelXDimension = 640;  
    PixelYDimension = 480;  
    SceneType = 1;  
    SensingMethod = 2;  
    ShutterSpeedValue = "4.584985584026477";  
    WhiteBalance = 0;  
}
```

```
#import <AVFoundation/AVFoundation.h>
#import "THBaseCameraController.h"

@protocol THTextureDelegate <NSObject>
- (void)textureCreatedWithTarget:(GLenum)target name:(GLuint)name;
@end

@interface THCameraController : THBaseCameraController

- (instancetype)initWithContext:(EAGLContext *)context;
@property (weak, nonatomic) id <THTextureDelegate> textureDelegate;

@end
```

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController () <AVCaptureVideoDataOutputSampleBufferDelegate>

@property (weak, nonatomic) EAGLContext *context;
@property (strong, nonatomic) AVCaptureVideoDataOutput *videoDataOutput;

@end

@implementation THCameraController

- (instancetype)initWithContext:(EAGLContext *)context {
    self = [super init];
    if (self) {
        _context = context;
    }
    return self;
}

- (NSString *)sessionPreset { // 1
    return AVCaptureSessionPreset640x480;
}

- (BOOL)setupSessionOutputs:(NSError **)error {

    self.videoDataOutput = [[AVCaptureVideoDataOutput alloc] init];
```

```
self.videoDataOutput.videoSettings = // 2
@{(id)kCVPixelBufferPixelFormatTypeKey : @(kCVPixelFormatType_32BGRA)};
```

```
[self.videoDataOutput setSampleBufferDelegate:self // 3
queue:dispatch_get_main_queue()];
```

```
if ([self.captureSession canAddOutput:self.videoDataOutput]) { // 4
    [self.captureSession addOutput:self.videoDataOutput];
    return YES;
}
```

```
return NO;
}
```

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
fromConnection:(AVCaptureConnection *)connection {
}
```

```
@end
```

```
@interface THCameraController () <AVCaptureVideoDataOutputSampleBufferDelegate>

@property (weak, nonatomic) EAGLContext *context;
@property (strong, nonatomic) AVCaptureVideoDataOutput *videoDataOutput;

@property (nonatomic) CVOpenGLTextureCacheRef textureCache; // 1
@property (nonatomic) CVOpenGLTextureRef cameraTexture;

@end

@implementation THCameraController

- (instancetype)initWithContext:(EAGLContext *)context {
    self = [super init];
    if (self) {
        _context = context;
        CVReturn err = CVOpenGLTextureCacheCreate(kCFAllocatorDefault, // 2
                                                NULL,
                                                _context,
                                                NULL,
                                                &_textureCache);
        if (err != kCVReturnSuccess) { // 3
            NSLog(@"Error creating texture cache. %d", err);
        }
    }
    return self;
}

}
```

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
fromConnection:(AVCaptureConnection *)connection {
    CVReturn err;
    CVImageBufferRef pixelBuffer = // 1
        CMSampleBufferGetImageBuffer(sampleBuffer);

    CMFormatDescriptionRef formatDescription = // 2
        CMSampleBufferGetFormatDescription(sampleBuffer);
    CMVideoDimensions dimensions =
        CMVideoFormatDescriptionGetDimensions(formatDescription);

    err = CVOpenGLTextureCacheCreateTextureFromImage(kCFAllocatorDefault, // 3
                                                    _textureCache,
                                                    pixelBuffer,
                                                    NULL,
                                                    GL_TEXTURE_2D,
                                                    GL_RGBA,
                                                    dimensions.height,
                                                    dimensions.height,
                                                    GL_BGRA,
                                                    GL_UNSIGNED_BYTE,
                                                    0,
                                                    &_cameraTexture);

    if (!err) {
        GLenum target = CVOpenGLTextureGetTarget(_cameraTexture); // 4
        GLuint name = CVOpenGLTextureGetName(_cameraTexture);
        [self.textureDelegate textureCreatedWithTarget:target name:name]; // 5
    } else {
        NSLog(@"Error at CVOpenGLTextureCacheCreateTextureFromImage %d", err);
    }

    [self cleanupTextures];
}

- (void)cleanupTextures { // 6
    if (_cameraTexture) {
        CFRelease(_cameraTexture);
        _cameraTexture = NULL;
    }
    CVOpenGLTextureCacheFlush(_textureCache, 0);
}
```

```
AVAsset *asset = // Asynchronously loaded video asset
AVAssetTrack *track =
    [[asset tracksWithMediaType:AVMediaTypeVideo] firstObject];

self.assetReader =
    [[AVAssetReader alloc] initWithAsset:asset error:nil];

NSDictionary *readerOutputSettings = @{
    (id)kCVPixelBufferPixelFormatTypeKey : @(kCVPixelFormatType_32BGRA)
};

AVAssetReaderTrackOutput *trackOutput =
    [[AVAssetReaderTrackOutput alloc] initWithTrack:track
                                         outputSettings:readerOutputSettings];

[self.assetReader addOutput:trackOutput];

[self.assetReader startReading];
```

```
NSURL *outputURL = // Destination output URL

self.assetWriter = [[AVAssetWriter alloc] initWithURL:outputURL
                                         fileType:AVFileTypeQuickTimeMovie
                                         error:nil];

NSDictionary *writerOutputSettings = @{
    AVVideoCodecKey: AVVideoCodecH264,
    AVVideoWidthKey: @1280,
    AVVideoHeightKey: @720,
    AVVideoCompressionPropertiesKey: @{
        AVVideoMaxKeyFrameIntervalKey: @1,
        AVVideoAverageBitRateKey: @10500000,
        AVVideoProfileLevelKey: AVVideoProfileLevelH264Main31,
    }
};

AVAssetWriterInput *writerInput =
[[AVAssetWriterInput alloc] initWithMediaType:AVMediaTypeVideo
                                outputSettings:writerOutputSettings];

[self.assetWriter addInput:writerInput];

[self.assetWriter startWriting];
```

```

// Serial Queue
dispatch_queue_t dispatchQueue =
    dispatch_queue_create("com.tapharmonic.WriterQueue", NULL);

[self.assetWriter startSessionAtSourceTime:kCMTimeZero];
[writerInput requestMediaDataWhenReadyOnQueue:dispatchQueue usingBlock:^{
    BOOL complete = NO;

    while ([writerInput isReadyForMoreMediaData] && !complete) {

        CMSampleBufferRef sampleBuffer = [trackOutput copyNextSampleBuffer];

        if (sampleBuffer) {
            BOOL result = [writerInput appendSampleBuffer:sampleBuffer];
            CFRelease(sampleBuffer);
            complete = !result;
        } else {
            [writerInput markAsFinished];
            complete = YES;
        }
    }

    if (complete) {
        [self.assetWriter finishWritingWithCompletionHandler:^{
            AVAssetWriterStatus status = self.assetWriter.status;
            if (status == AVAssetWriterStatusCompleted) {
                // Handle success case
            } else {
                // Handle failure case
            }
        }];
    }
}];
}

```

```
#import <AVFoundation/AVFoundation.h>

typedef void(^THSampleDataCompletionBlock) (NSData *);

@interface THSampleDataProvider : NSObject

+ (void)loadAudioSamplesFromAsset:(AVAsset *)asset
                           completionBlock:(THSampleDataCompletionBlock)completionBlock;

@end
```

```
#import "THSampleDataProvider.h"

@implementation THSampleDataProvider

+ (void)loadAudioSamplesFromAsset:(AVAsset *)asset
    completionBlock:(THSampleDataCompletionBlock)completionBlock {

    NSString *tracks = @"tracks";

    [asset loadValuesAsynchronouslyForKeys:@[tracks] completionHandler:^{
        // 1

        AVKeyValueStatus status = [asset statusOfValueForKey:tracks error:nil];

        NSData *sampleData = nil;

        if (status == AVKeyValueStatusLoaded) { // 2
            sampleData = [self readAudioSamplesFromAsset:asset];
        }

        dispatch_async(dispatch_get_main_queue(), ^{
            // 3
            completionBlock(sampleData);
        });
    }];
}

+ (NSData *)readAudioSamplesFromAsset:(AVAsset *)asset {
    // To be implemented

    return nil;
}

@end
```

```
+ (NSData *)readAudioSamplesFromAsset:(AVAsset *)asset {

    NSError *error = nil;

    AVAssetReader *assetReader = // 1
        [[AVAssetReader alloc] initWithAsset:asset error:&error];

    if (!assetReader) {
        NSLog(@"Error creating asset reader: %@", [error localizedDescription]);
        return nil;
    }

    AVAssetTrack *track = // 2
        [[asset tracksWithMediaType:AVMediaTypeAudio] firstObject];

    NSDictionary *outputSettings = @{ // 3
        AVFormatIDKey : @(kAudioFormatLinearPCM),
        AVLinearPCMIsBigEndianKey : @NO,
        AVLinearPCMIsFloatKey : @NO,
        AVLinearPCMBitDepthKey : @(16)
    };

    AVAssetReaderTrackOutput *trackOutput = // 4
        [[AVAssetReaderTrackOutput alloc] initWithTrack:track
                                             outputSettings:outputSettings];

    [assetReader addOutput:trackOutput];
}
```

```
[assetReader startReading];

NSMutableData *sampleData = [NSMutableData data];

while (assetReader.status == AVAssetReaderStatusReading) {

    CMSampleBufferRef sampleBuffer = [trackOutput copyNextSampleBuffer];// 5

    if (sampleBuffer) {

        CMBlockBufferRef blockBufferRef = // 6
            CMSampleBufferGetDataBuffer(sampleBuffer);

        size_t length = CMBlockBufferGetDataLength(blockBufferRef);
        SInt16 sampleBytes[length];

        CMBlockBufferCopyDataBytes(blockBufferRef, // 7
                                   0,
                                   length,
                                   sampleBytes);

        [sampleData appendBytes:sampleBytes length:length];

        CMSampleBufferInvalidate(sampleBuffer); // 8
        CFRelease(sampleBuffer);
    }
}

if (assetReader.status == AVAssetReaderStatusCompleted) { // 9
    return sampleData;
} else {
    NSLog(@"Failed to read audio samples from asset");
    return nil;
}
}
```

```
@interface THSampleDataFilter : NSObject  
  
- (id)initWithData:(NSData *)sampleData;  
  
- (NSArray *)filteredSamplesForSize:(CGSize)size;  
  
@end
```

```
#import "THSampleDataFilter.h"

@interface THSampleDataFilter ()
@property (nonatomic, strong) NSData *sampleData;
@end

@implementation THSampleDataFilter

- (id)initWithData:(NSData *)sampleData {
    self = [super init];
    if (self) {
        _sampleData = sampleData;
    }
    return self;
}

- (NSArray *)filteredSamplesForSize:(CGSize)size {
    NSMutableArray *filteredSamples = [[NSMutableArray alloc] init]; // 1
    NSUInteger sampleCount = self.sampleData.length / sizeof(SInt16);
    NSUInteger binSize = sampleCount / size.width;

    SInt16 *bytes = (SInt16 *)self.sampleData.bytes;
    SInt16 maxSample = 0;
```

```

for (NSUInteger i = 0; i < sampleCount; i += binSize) {

    SInt16 sampleBin[binSize];

    for (NSUInteger j = 0; j < binSize; j++) { // 2
        sampleBin[j] = CFSwapInt16LittleToHost(bytes[i + j]);
    }

    SInt16 value = [self maxValueInArray:sampleBin ofSize:binSize]; // 3
    [filteredSamples addObject:@(value)];

    if (value > maxSample) { // 4
        maxSample = value;
    }
}

CGFloat scaleFactor = (size.height / 2) / maxSample;

for (NSUInteger i = 0; i < filteredSamples.count; i++) { // 5
    filteredSamples[i] = @([filteredSamples[i] integerValue] * scaleFactor);
}

return filteredSamples;
}

- (SInt16)maxValueInArray:(SInt16 [])values ofSize:(NSUInteger)size {
    SInt16 maxValue = 0;
    for (int i = 0; i < size; i++) {
        if (abs(values[i]) > maxValue) {
            maxValue = abs(values[i]);
        }
    }
    return maxValue;
}

@end

```

```
@class AVAsset;

@interface THWaveformView : UIView

@property (strong, nonatomic) AVAsset *asset;
@property (strong, nonatomic) UIColor *waveColor;

@end
```

```
#import "THWaveformView.h"
#import "THSampleDataProvider.h"
#import "THSampleDataFilter.h"
#import <QuartzCore/QuartzCore.h>

static const CGFloat THWidthScaling = 0.95;
static const CGFloat THHeightScaling = 0.85;

@interface THWaveformView ()
@property (strong, nonatomic) THSampleDataFilter *filter;
@property (strong, nonatomic) UIActivityIndicatorView *loadingView;
@end

@implementation THWaveformView

...

- (void)setAsset:(AVAsset *)asset {
    if (_asset != asset) {
        _asset = asset;

        [THSampleDataProvider loadAudioSamplesFromAsset:self.asset           // 1
         completionBlock:^(NSData *sampleData) {

            self.filter =
                [[THSampleDataFilter alloc] initWithData:sampleData];
            // 2

            [self.loadingView stopAnimating];                                // 3
            [self setNeedsDisplay];
        }];
    }
}

- (void)drawRect:(CGRect)rect {
    // To be implemented
}

@end
```

```
- (void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextScaleCTM(context, THWidthScaling, THHeightScaling); // 1

    CGFloat xOffset = self.bounds.size.width -
        (self.bounds.size.width * THWidthScaling);

    CGFloat yOffset = self.bounds.size.height -
        (self.bounds.size.height * THHeightScaling);

    CGContextTranslateCTM(context, xOffset / 2, yOffset / 2);

    NSArray *filteredSamples = // 2
        [self.filter filteredSamplesForSize:self.bounds.size];

    CGFloat midY = CGRectGetMidY(rect);

    CGMutablePathRef halfPath = CGPathCreateMutable(); // 3
    CGPathMoveToPoint(halfPath, NULL, 0.0f, midY);

    for (NSUInteger i = 0; i < filteredSamples.count; i++) {
        float sample = [filteredSamples[i] floatValue];
        CGPathAddLineToPoint(halfPath, NULL, i, midY - sample);
    }

    CGPathAddLineToPoint(halfPath, NULL, filteredSamples.count, midY);

    CGMutablePathRef fullPath = CGPathCreateMutable(); // 4
    CGPathAddPath(fullPath, NULL, halfPath);

    CGAffineTransform transform = CGAffineTransformIdentity; // 5
    transform = CGAffineTransformTranslate(transform, 0, CGRectGetHeight(rect));
    transform = CGAffineTransformScale(transform, 1.0, -1.0);
    CGPathAddPath(fullPath, &transform, halfPath);

    CGContextAddPath(context, fullPath); // 6
    CGContextSetFillColorWithColor(context, self.waveColor.CGColor);
    CGContextDrawPath(context, kCGPathFill);

    CGPathRelease(halfPath); // 7
    CGPathRelease(fullPath);
}
```

```
#import "THImageTarget.h"
#import "THBaseCameraController.h"

@interface THCameraController : THBaseCameraController

- (void)startRecording;
- (void)stopRecording;
@property (nonatomic, getter = isRecording) BOOL recording;

@property (weak, nonatomic) id <THImageTarget> imageTarget;

@end
```

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>

@interface THCameraController () <AVCaptureVideoDataOutputSampleBufferDelegate,
AVCaptureAudioDataOutputSampleBufferDelegate>

@property (strong, nonatomic) AVCaptureVideoDataOutput *videoDataOutput;
@property (strong, nonatomic) AVCaptureAudioDataOutput *audioDataOutput;

@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError **)error {

    self.videoDataOutput = [[AVCaptureVideoDataOutput alloc] init];           // 1

    NSDictionary *outputSettings =
    @{@"(id)kCVPixelBufferPixelFormatTypeKey : @(kCVPixelFormatType_32BGRA)};

    self.videoDataOutput.videoSettings = outputSettings;
    self.videoDataOutput.alwaysDiscardsLateVideoFrames = NO;                   // 2

    [self.videoDataOutput setSampleBufferDelegate:self
                           queue:self.dispatchQueue];
    if ([self.captureSession canAddOutput:self.videoDataOutput]) {
        [self.captureSession addOutput:self.videoDataOutput];
    } else {
        return NO;
    }

    self.audioDataOutput = [[AVCaptureAudioDataOutput alloc] init];           // 3

    [self.audioDataOutput setSampleBufferDelegate:self
                           queue:self.dispatchQueue];

    if ([self.captureSession canAddOutput:self.audioDataOutput]) {
        [self.captureSession addOutput:self.audioDataOutput];
    } else {
        return NO;
    }

    return YES;
}

- (NSString *)sessionPreset {                                              // 4
    return AVCaptureSessionPresetMedium;
```

```
}

- (void)startRecording {
    // To be implemented
}

- (void)stopRecording {
    // To be implemented
}

#pragma mark - Delegate methods

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
fromConnection:(AVCaptureConnection *)connection {

    // To be implemented
}

@end
```

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
fromConnection:(AVCaptureConnection *)connection {
    if (captureOutput == self.videoDataOutput) { // 1
        CVPixelBufferRef imageBuffer =
            CMSampleBufferGetImageBuffer(sampleBuffer); // 2

        CIImage *sourceImage =
            [CIImage imageWithCVPixelBuffer:imageBuffer options:nil]; // 3

        [self.imageTarget setImage:sourceImage];
    }
}
```

```
#import <AVFoundation/AVFoundation.h>

@protocol THMovieWriterDelegate <NSObject>
- (void)didWriteMovieAtURL:(NSURL *)outputURL;
@end

@interface THMovieWriter : NSObject

- (id)initWithVideoSettings:(NSDictionary *)videoSettings           // 1
    audioSettings:(NSDictionary *)audioSettings
    dispatchQueue:(dispatch_queue_t)dispatchQueue;

- (void)startWriting;
- (void)stopWriting;
@property (nonatomic) BOOL isWriting;

@property (weak, nonatomic) id<THMovieWriterDelegate> delegate;          // 2

- (void)processSampleBuffer:(CMSSampleBufferRef)sampleBuffer;           // 3

@end
```

```
#import "THMovieWriter.h"
#import <AVFoundation/AVFoundation.h>
#import "THContextManager.h"
#import "THFunctions.h"
#import "THPhotoFilters.h"
#import "THNotifications.h"

static NSString *const THVideoFilename = @"movie.mov";

@interface THMovieWriter {}

@property (strong, nonatomic) AVAssetWriter *assetWriter; // 1
@property (strong, nonatomic) AVAssetWriterInput *assetWriterVideoInput;
@property (strong, nonatomic) AVAssetWriterInput *assetWriterAudioInput;
@property (strong, nonatomic)
    AVAssetWriterInputPixelBufferAdaptor *assetWriterInputPixelBufferAdaptor;

@property (strong, nonatomic) dispatch_queue_t dispatchQueue;

@property (weak, nonatomic) CIContext *ciContext;
@property (nonatomic) CGColorSpaceRef colorSpace;
@property (strong, nonatomic) CIFilter *activeFilter;

@property (strong, nonatomic) NSDictionary *videoSettings;
@property (strong, nonatomic) NSDictionary *audioSettings;

@property (nonatomic) BOOL firstSample;
```

```

@end

@implementation THMovieWriter

- (id)initWithVideoSettings:(NSDictionary *)videoSettings
    audioSettings:(NSDictionary *)audioSettings
    dispatchQueue:(dispatch_queue_t)dispatchQueue {
    self = [super init];
    if (self) {
        _videoSettings = videoSettings;
        _audioSettings = audioSettings;
        _dispatchQueue = dispatchQueue;

        _ciContext = [THContextManager sharedInstance].ciContext;           // 2
        _colorSpace = CGColorSpaceCreateDeviceRGB();

        _activeFilter = [THPhotoFilters defaultFilter];
        _firstSample = YES;
    }

    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];      // 3
    [nc addObserver:self
        selector:@selector(filterChanged:)
        name:THFilterSelectionChangedNotification
        object:nil];
}

return self;
}

- (void)dealloc {
    CGColorSpaceRelease(_colorSpace);
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}

- (void)filterChanged:(NSNotification *)notification {
    self.activeFilter = [notification.object copy];
}

- (void)startWriting {
    // To be implemented
}

```

```
- (void)processSampleBuffer:(CMSSampleBufferRef)sampleBuffer
                      mediaType:(CMMediaType)mediaType {
    // To be implemented
}

- (void)stopWriting {
    // To be implemented
}

- (NSURL *)outputURL {                                // 4
    NSString *filePath =
        [NSTemporaryDirectory() stringByAppendingPathComponent:THVideoFilename];
    NSURL *url = [NSURL fileURLWithPath:filePath];
    if ([[NSFileManager defaultManager] fileExistsAtPath:url.path]) {
        [[NSFileManager defaultManager] removeItemAtURL:url error:nil];
    }
    return url;
}

@end
```

```
- (void)startWriting {
    dispatch_async(self.dispatchQueue, ^{
        // 1

        NSError *error = nil;

        NSString *fileType = AVFileTypeQuickTimeMovie;
        self.assetWriter =
            [AVAssetWriter assetWriterWithURL:[self outputURL]
                fileType:fileType
                error:&error];
        if (!self.assetWriter || error) {
            NSString *formatString = @"Could not create AVAssetWriter: %@";
            NSLog(@"%@", [NSString stringWithFormat:formatString, error]);
            return;
        }

        self.assetWriterVideoInput =
            [[AVAssetWriterInput alloc] initWithMediaType:AVMediaTypeVideo
                outputSettings:self.videoSettings];
        self.assetWriterVideoInput.expectsMediaDataInRealTime = YES;

        UIDeviceOrientation orientation = [UIDevice currentDevice].orientation;
        self.assetWriterVideoInput.transform =
            // 4
            THTransformForDeviceOrientation(orientation);
```

```

NSDictionary *attributes = @{
                           // 5
                           (id)kCVPixelBufferPixelFormatTypeKey : @(kCVPixelFormatType_32BGRA),
                           (id)kCVPixelBufferWidthKey : self.videoSettings[AVVideoWidthKey],
                           (id)kCVPixelBufferHeightKey : self.videoSettings[AVVideoHeightKey],
                           (id)kCVPixelFormatOpenGLESCompatibility : (id)kCFBooleanTrue
};

self.assetWriterInputPixelBufferAdaptor =
// 6
[[AVAssetWriterInputPixelBufferAdaptor alloc]
 initWithAssetWriterInput:self.assetWriterVideoInput
 sourcePixelBufferAttributes:attributes];

if ([self.assetWriter canAddInput:self.assetWriterVideoInput]) { // 7
    [self.assetWriter addInput:self.assetWriterVideoInput];
} else {
    NSLog(@"Unable to add video input.");
    return;
}

self.assetWriterAudioInput =
// 8
[[AVAssetWriterInput alloc] initWithMediaType:AVMediaTypeAudio
                                     outputSettings:self.audioSettings];

self.assetWriterAudioInput.expectsMediaDataInRealTime = YES;

if ([self.assetWriter canAddInput:self.assetWriterAudioInput]) { // 9
    [self.assetWriter addInput:self.assetWriterAudioInput];
} else {
    NSLog(@"Unable to add audio input.");
    return;
}

self.isWriting = YES; // 10
self.firstSample = YES;
});

}

```

```
- (void)processSampleBuffer:(CMSSampleBufferRef)sampleBuffer {

    if (!self.isWriting) {
        return;
    }

    CMFormatDescriptionRef formatDesc = // 1
        CMSampleBufferGetFormatDescription(sampleBuffer);

    CMMediaType mediaType = CMFormatDescriptionGetMediaType(formatDesc);

    if (mediaType == kCMMediaType_Video) {

        CMTime timestamp =
            CMSampleBufferGetPresentationTimeStamp(sampleBuffer);

        if (self.firstSample) { // 2
            if ([self.assetWriter startWriting]) {
                [self.assetWriter startSessionAtSourceTime:timestamp];
            } else {
                NSLog(@"Failed to start writing.");
            }
            self.firstSample = NO;
        }
    }
}
```

```
CVPixelBufferRef outputRenderBuffer = NULL;

CVPixelBufferPoolRef pixelBufferPool =
    self.assetWriterInputPixelBufferAdaptor.pixelBufferPool;

OSStatus err = CVPixelBufferPoolCreatePixelBuffer(NULL,           // 3
                                                pixelBufferPool,
                                                &outputRenderBuffer);

if (err) {
    NSLog(@"Unable to obtain a pixel buffer from the pool.");
    return;
}

CVPixelBufferRef imageBuffer =                                     // 4
    CMSampleBufferGetImageBuffer(sampleBuffer);

CIIImage *sourceImage = [CIIImage imageWithCVPixelBuffer:imageBuffer
                           options:nil];

[self.activeFilter setValue:sourceImage forKey:kCIInputImageKey];

CIIImage *filteredImage = self.activeFilter.outputImage;

if (!filteredImage) {
    filteredImage = sourceImage;
```

```
}

    [self.ciContext render:filteredImage           // 5
     toCVPixelBuffer:outputRenderBuffer
      bounds:filteredImage.extent
     colorSpace:self.colorSpace];

if (self.assetWriterVideoInput.readyForMoreMediaData) {           // 6
    if (! [self.assetWriterInputPixelBufferAdaptor
            appendPixelBuffer:outputRenderBuffer
            withPresentationTime:timestamp]) {
        NSLog(@"Error appending pixel buffer.");
    }
}

CVPixelBufferRelease(outputRenderBuffer);

}

else if (!self.firstSample && mediaType == kCMMediaType_Audio) {      // 7
    if (self.assetWriterAudioInput.isReadyForMoreMediaData) {
        if (! [self.assetWriterAudioInput appendSampleBuffer:sampleBuffer]) {
            NSLog(@"Error appending audio sample buffer.");
        }
    }
}
}
```

```
- (void)stopWriting {
    self.isWriting = NO;                                     // 1

    dispatch_async(self.dispatchQueue, ^{
        [self.assetWriter finishWritingWithCompletionHandler:^{
            if (self.assetWriter.status == AVAssetWriterStatusCompleted) {
                dispatch_async(dispatch_get_main_queue(), ^{
                    NSURL *fileURL = [self.assetWriter outputURL];
                    [self.delegate didWriteMovieAtURL:fileURL];
                });
            } else {
                NSLog(@"Failed to write movie: %@", self.assetWriter.error);
            }
        }];
    });
}
```

```
#import "THCameraController.h"
#import <AVFoundation/AVFoundation.h>
#import "THMovieWriter.h"
#import <AssetsLibrary/AssetsLibrary.h>

@interface THCameraController () <AVCaptureVideoDataOutputSampleBufferDelegate,
    AVCaptureAudioDataOutputSampleBufferDelegate,
    THMovieWriterDelegate>

@property (strong, nonatomic) AVCaptureVideoDataOutput *videoDataOutput;
@property (strong, nonatomic) AVCaptureAudioDataOutput *audioDataOutput;

@property (strong, nonatomic) THMovieWriter *movieWriter; // 1

@end

@implementation THCameraController

- (BOOL)setupSessionOutputs:(NSError **)error {

    // AVCaptureVideoDataOutput and AVCaptureAudioDataOutput set up and
    // configuration previously covered in Listing 8.x

    NSString *fileType = AVFileTypeQuickTimeMovie;

    NSDictionary *videoSettings =
        [self.videoDataOutput
            recommendedVideoSettingsForAssetWriterWithOutputFileType:fileType]; // 2
```

```
    NSDictionary *audioSettings =
        [self.audioDataOutput
            recommendedAudioSettingsForAssetWriterWithOutputFileType:fileType];

    self.movieWriter = // 3
        [[THMovieWriter alloc] initWithVideoSettings:videoSettings
                                            audioSettings:audioSettings
                                              dispatchQueue:self.dispatchQueue];
    self.movieWriter.delegate = self;

    return YES;
}

- (NSString *)sessionPreset {
    return AVCaptureSessionPreset1280x720;
}

- (void)startRecording { // 4
    [self.movieWriter startWriting];
    self.recording = YES;
}
```

```

- (void)stopRecording {
    [self.movieWriter stopWriting];
    self.recording = NO;
}

#pragma mark - Delegate methods

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
fromConnection:(AVCaptureConnection *)connection {

    [self.movieWriter processSampleBuffer:sampleBuffer]; // 5

    if (captureOutput == self.videoDataOutput) {

        CVPixelBufferRef imageBuffer =
            CMSampleBufferGetImageBuffer(sampleBuffer);

        CIImage *sourceImage =
            [CIImage imageWithCVPixelBuffer:imageBuffer options:nil];

        [self.imageTarget setImage:sourceImage];
    }
}

- (void)didWriteMovieAtURL:(NSURL *)outputURL { // 6
    ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];

    if ([library videoAtPathIsCompatibleWithSavedPhotosAlbum:outputURL]) {

        ALAssetsLibraryWriteVideoCompletionBlock completionBlock;

        completionBlock = ^(NSURL *assetURL, NSError *error){
            if (error) {
                [self.delegate assetLibraryWriteFailedWithError:error];
            }
        };

        [library writeVideoAtPathToSavedPhotosAlbum:outputURL
                                         completionBlock:completionBlock];
    }
}

@end

```

```
typedef struct {
    CMTimeValue value;
    CMTimeScale timescale;
    CMTimeFlags flags;
    CMTimeEpoch epoch;
} CMTime;
```

```
CMTIME t1 = CMTIMEMake(3, 1);
CMTIME t2 = CMTIMEMake(1800, 600);
CMTIME t3 = CMTIMEMake(3000, 1000);
CMTIME t4 = CMTIMEMake(132300, 44100);
```

```
CMTTimeShow(t1) ; // --> {3/1 = 3.000}
CMTTimeShow(t2) ; // --> {1800/600 = 3.000}
CMTTimeShow(t3) ; // --> {3000/1000 = 3.000}
CMTTimeShow(t4) ; // --> {132300/44100 = 3.000}
```

```
CMTIME time1 = CMTIMEMake(5, 1);
CMTIME time2 = CMTIMEMake(3, 1);

CMTIME result;

result = CMTIMEAdd(time1, time2);
CMTIMEShow(result); // --> {8/1 = 8.000}

result = CMTIMESubtract(time1, time2);
CMTIMEShow(result); // --> {2/1 = 2.000}
```

```
CMTIME time = CMTIMEMake(5000, 1000);

CMTIME doubledTime = CMTIMEMake(time.value, time.timescale / 2);
CMTIMEShow(doubledTime); // --> {5000/500 = 10.000}

CMTIME halvedTime = CMTIMEMake(time.value, time.timescale * 2);
CMTIMEShow(halvedTime); // --> {5000/2000 = 2.500}
```

```
CMTIME fiveSecondsTime = CMTIMEMake(5, 1);
CMTIMERange timeRange = CMTIMERangeMake(fiveSecondsTime, fiveSecondsTime);
CMTIMERangeShow(timeRange); // --> {{5/1 = 5.000}, {5/1 = 5.000}}
```

```
CMTIME fiveSeconds = CMTIMEMake(5, 1);
CMTIME tenSeconds = CMTIMEMake(10, 1);
CMTIMERange timeRange = CMTIMERangeFromTimeToTime(fiveSeconds, tenSeconds);
CMTIMERangeShow(timeRange); // --> {{5/1 = 5.000}, {5/1 = 5.000}}
```

```
CMTTimeRange range1 = CMTTimeRangeMake(kCMTTimeZero, CMTTimeMake(5, 1));
CMTTimeRange range2 = CMTTimeRangeMake(CMTTimeMake(2, 1), CMTTimeMake(5, 1));

CMTTimeRange intersectionRange = CMTTimeRangeGetIntersection(range1, range2);
CMTTimeRangeShow(intersectionRange); // --> {{2/1 = 2.000}, {3/1 = 3.000} }

CMTTimeRange unionRange = CMTTimeRangeGetUnion(range1, range2);
CMTTimeRangeShow(unionRange); // --> {{0/1 = 0.000}, {7/1 = 7.000} }
```

```
NSURL *url =
[[NSBundle mainBundle] URLForResource:@"video" withExtension:@"mp4"] ;

NSDictionary *options = @{@"AVURLAssetPreferPreciseDurationAndTimingKey" : @YES};
AVAsset *asset = [AVURLAsset URLAssetWithURL:url options:options];

// Asset "keys" to load
NSArray *keys = @[@"tracks", @"duration", @"commonMetadata"] ;

[asset loadValuesAsynchronouslyForKeys:keys completionHandler:^{
    // Validate loaded status of keys
}];
```

```
AVAsset *goldenGateAsset = // prepared golden gate asset
AVAsset *teaGardenAsset = // prepared tea garden asset
AVAsset *soundtrackAsset = // prepared sound track asset

AVMutableComposition *composition = [AVMutableComposition composition];

// Video Track
AVMutableCompositionTrack *videoTrack =
    [composition addMutableTrackWithMediaType:AVMediaTypeVideo
        preferredTrackID:kCMPersistentTrackID_Invalid];
// Audio Track
AVMutableCompositionTrack *audioTrack =
    [composition addMutableTrackWithMediaType:AVMediaTypeAudio
        preferredTrackID:kCMPersistentTrackID_Invalid];
```

```

// The "insertion cursor" time
CMTime cursorTime = kCMTimeZero; // 1

CMTime videoDuration = CMTimeMake(5, 1); // 2
CMTimeRange videoTimeRange = CMTimeRangeMake(kCMTimeZero, videoDuration);

AVAssetTrack *assetTrack;

// Extract and insert Golden Gate Segment
assetTrack = // 3
    [[goldenGateAsset tracksWithMediaType:AVMediaTypeVideo] firstObject];
[videoTrack insertTimeRange:videoTimeRange
    ofTrack:assetTrack
    atTime:cursorTime error:nil];

// Increment cursor time
cursorTime = CMTimeAdd(cursorTime, videoDuration); // 4


// Extract and insert Tea Garden segment
assetTrack = // 5
    [[teaGardenAsset tracksWithMediaType:AVMediaTypeVideo] firstObject];
[videoTrack insertTimeRange:videoTimeRange
    ofTrack:assetTrack
    atTime:cursorTime error:nil];

// Reset cursor time
cursorTime = kCMTimeZero; // 6
CMTime audioDuration = composition.duration;
CMTimeRange audioTimeRange = CMTimeRangeMake(kCMTimeZero, audioDuration);

// Extract and insert Tea Garden segment
assetTrack = // 7
    [[soundtrackAsset tracksWithMediaType:AVMediaTypeAudio] firstObject];
[audioTrack insertTimeRange:audioTimeRange
    ofTrack:assetTrack
    atTime:cursorTime error:nil];

```

```
#import <AVFoundation/AVFoundation.h>

@protocol THComposition <NSObject>

- (AVPlayerItem *)makePlayable;
- (AVAssetExportSession *)makeExportable;

@end
```

```
#import <AVFoundation/AVFoundation.h>
#import "THComposition.h"

@interface THBasicComposition : NSObject <THComposition>

@property (strong, readonly, nonatomic) AVComposition *composition;

+ (instancetype)compositionWithComposition:(AVComposition *)composition;
- (instancetype)initWithComposition:(AVComposition *)composition;

@end
```

```
#import <AVFoundation/AVFoundation.h>
#import "THComposition.h"

@protocol THCompositionBuilder <NSObject>

- (id <THComposition>)buildComposition;

@end
```

```
#import "THCompositionBuilder.h"
#import "THTimeline.h"

@interface THBasicCompositionBuilder : NSObject <THCompositionBuilder>

- (id)initWithTimeline:(THTimeline *)timeline;

@end
```

```
#import "THBasicCompositionBuilder.h"
#import "THBasicComposition.h"
#import "THFunctions.h"

@interface THBasicCompositionBuilder ()
@property (strong, nonatomic) THTimeline *timeline;
@property (strong, nonatomic) AVMutableComposition *composition;
@end

@implementation THBasicCompositionBuilder

- (id)initWithTimeline:(THTimeline *)timeline {
    self = [super init];
    if (self) {
        _timeline = timeline;
    }
    return self;
}

- (id <THComposition>)buildComposition {

    self.composition = [AVMutableComposition composition]; // 1

    [self addCompositionTrackOfType:AVMediaTypeVideo
                           withMediaItems:self.timeline.videos];

    [self addCompositionTrackOfType:AVMediaTypeAudio
                           withMediaItems:self.timeline.voiceOvers];

    [self addCompositionTrackOfType:AVMediaTypeAudio
                           withMediaItems:self.timeline.musicItems];

    // Create and return the basic composition // 2
    return [THBasicComposition compositionWithComposition:self.composition];
}

- (void)addCompositionTrackOfType:(NSString *)mediaType
                           withMediaItems:(NSArray *)mediaItems {

    // To be implemented
}

@end
```

```
@implementation THBasicCompositionBuilder

...
- (void)addCompositionTrackOfType:(NSString *)mediaType
    withMediaItems:(NSArray *)mediaItems {

    if (!THIsEmpty(mediaItems)) { // 1

        CMPersistentTrackID trackID = kCMPersistentTrackID_Invalid;

        AVMutableCompositionTrack *compositionTrack = // 2
            [self.composition addMutableTrackWithMediaType:mediaType
                preferredTrackID:trackID];

        // Set insert cursor to 0
        CMTime cursorTime = kCMTimeZero; // 3

        for (THMediaItem *item in mediaItems) {

            if (CMTIME_COMPARE_INLINE(item.startTimeInTimeline,
                !=,
                kCMTimeInvalid)) { // 4
                cursorTime = item.startTimeInTimeline;
            }

            AVAssetTrack *assetTrack = // 5
                [[item.asset tracksWithMediaType:mediaType] firstObject];

            [compositionTrack insertTimeRange:item.timeRange // 6
                ofTrack:assetTrack
                atTime:cursorTime
                error:nil];
        }

        // Move cursor to next item time
        cursorTime = CMTimeAdd(cursorTime, item.timeRange.duration); // 7
    }
}

@end
```

```
#import "THTimeline.h"
#import "THComposition.h"

@interface THCompositionExporter : NSObject

@property (nonatomic) BOOL exporting;
@property (nonatomic) CGFloat progress;

- (instancetype)initWithComposition:(id <THComposition>)composition;
- (void)beginExport;

@end
```

```
#import "THCompositionExporter.h"
#import "UIAlertView+THAdditions.h"
#import <AssetsLibrary/AssetsLibrary.h>

@interface THCompositionExporter ()
@property (strong, nonatomic) id <THComposition> composition;
@property (strong, nonatomic) AVAssetExportSession *exportSession;
@end

@implementation THCompositionExporter

- (instancetype)initWithComposition:(id <THComposition>)composition {
    self = [super init];
    if (self) {
        _composition = composition;
    }
    return self;
}

- (void)beginExport {
    self.exportSession = [self.composition makeExportable]; // 1
    self.exportSession.outputURL = [self exportURL];
    self.exportSession.outputFileType = AVFileTypeMPEG4;

    [self.exportSession exportAsynchronouslyWithCompletionHandler:^{ // 2
        // To be implemented
    }];
}
```

```
};

self.exporting = YES; // 3
[self monitorExportProgress]; // 4
}

- (void)monitorExportProgress {
    // To be implemented
}

- (NSURL *)exportURL {
    NSString *filePath = nil;
    NSUInteger count = 0;
    do {
        filePath = NSTemporaryDirectory();
        NSString *numberString = count > 0 ?
            [NSString stringWithFormat:@"-%li", (unsigned long) count] : @"";
        NSString *fileNameString =
            [NSString stringWithFormat:@"Masterpiece-%@.m4v", numberString];
        filePath = [filePath stringByAppendingPathComponent:fileNameString];
        count++;
    } while ([[NSFileManager defaultManager] fileExistsAtPath:filePath]);

    return [NSURL fileURLWithPath:filePath];
}

@end
```

```
@implementation THCompositionExporter

...
- (void)monitorExportProgress {
    double delayInSeconds = 0.1;
    int64_t delta = (int64_t)delayInSeconds * NSEC_PER_SEC;
    dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW, delta);

    dispatch_after(popTime, dispatch_get_main_queue(), ^{
        AVAssetExportSessionStatus status = self.exportSession.status;

        if (status == AVAssetExportSessionStatusExporting) { // 1

            self.progress = self.exportSession.progress;
            [self monitorExportProgress];

        } else {
            self.exporting = NO;
        }
    });
}

...
@end
```

```
@implementation THCompositionExporter

...
- (void)beginExport {
    self.exportSession = [self.composition makeExportable];
    self.exportSession.outputURL = [self exportURL];
    self.exportSession.outputFileType = AVFileTypeMPEG4;

    [self.exportSession exportAsynchronouslyWithCompletionHandler:^{
        dispatch_async(dispatch_get_main_queue(), ^{
            AVAssetExportSessionStatus status = self.exportSession.status;
            if (status == AVAssetExportSessionStatusCompleted) {
                [self writeExportedVideoToAssetsLibrary];
            } else {
                [UIAlertView showAlertWithTitle:@"Export Failed"
                                         message:@"The requested export failed."];
            }
        });
    }];
    self.exporting = YES;
    [self monitorExportProgress];
}
}
```

```
- (void)writeExportedVideoToAssetsLibrary {
    NSURL *exportURL = self.exportSession.outputURL;
    ALAssetsLibrary *library = [[ALAssetsLibrary alloc] init];

    if ([library videoAtPathIsCompatibleWithSavedPhotosAlbum:exportURL]) { // 2

        [library writeVideoAtPathToSavedPhotosAlbum:exportURL // 3
                                         completionBlock:^(NSURL *assetURL,
                                                       NSError *error) {

            if (error) { // 4
                NSString *message = @"Unable to write to Photos library.";
                [UIAlertView showAlertWithTitle:@"Write Failed"
                                         message:message];
            }
        }

        [[NSFileManager defaultManager] removeItemAtURL:exportURL
                                         error:nil];
    }];
} else {
    NSLog(@"Video could not be exported to the assets library.");
}
}

@end
```

```
AVCompositionTrack *track = // audio track in composition

// Define automation times
CMTime twoSeconds = CMTimeMake(2, 1);
CMTime fourSeconds = CMTimeMake(4, 1);
CMTime sevenSeconds = CMTimeMake(7, 1);

// Create a new parameters object for the given audio track
AVMutableAudioMixInputParameters *parameters =
    [AVMutableAudioMixInputParameters audioMixInputParametersWithTrack:track];

// Set initial volume
[parameters setVolume:0.5f atTime:kCMTimeZero];

// Define time range of the volume ramp
CMTimeRange range = CMTimeRangeFromTimeToTime(twoSeconds, fourSeconds);

// Perform 2 second ramp from 0.5 -> 0.8
[parameters setVolumeRampFromStartVolume:0.5f toEndVolume:0.8f timeRange:range];

// Drop volume to 0.3 at the 7-second mark
[parameters setVolume:0.3f atTime:sevenSeconds];

// Create a new audio mix instance
AVMutableAudioMix *audioMix = [AVMutableAudioMix audioMix];

// Assign the input parameters to the audio mix
audioMix.inputParameters = @[parameters];
```

```
#import "THComposition.h"

@interface THAudioMixComposition : NSObject <THComposition>

@property (strong, nonatomic, readonly) AVAudioMix *audioMix;
@property (strong, nonatomic, readonly) AVComposition *composition;

+ (instancetype)compositionWithComposition:(AVComposition *)composition
                                      audioMix:(AVAudioMix *)audioMix;

- (instancetype)initWithComposition:(AVComposition *)composition
                           audioMix:(AVAudioMix *)audioMix;

@end
```

```
#import "THAudioMixComposition.h"

@interface THAudioMixComposition ()
@property (strong, nonatomic) AVAudioMix *audioMix;
@property (strong, nonatomic) AVComposition *composition;
@end

@implementation THAudioMixComposition

+ (instancetype)compositionWithComposition:(AVComposition *)composition
                                      audioMix:(AVAudioMix *)audioMix {
    return [[self alloc] initWithComposition:composition audioMix:audioMix];
}

- (instancetype)initWithComposition:(AVComposition *)composition
                           audioMix:(AVAudioMix *)audioMix {
    self = [super init];
    if (self) {
        _composition = composition;
        _audioMix = audioMix;
    }
    return self;
}

- (AVPlayerItem *)makePlayable // 1
{
    AVPlayerItem *playerItem =
    [AVPlayerItem playerItemWithAsset:[self.composition copy]];
    playerItem.audioMix = self.audioMix;
    return playerItem;
}

- (AVAssetExportSession *)makeExportable // 2
{
    NSString *preset = AVAssetExportPresetHighestQuality;
    AVAssetExportSession *session =
    [AVAssetExportSession exportSessionWithAsset:[self.composition copy]
                                         presetName:preset];
    session.audioMix = self.audioMix;
    return session;
}

@end
```

```
#import "THCompositionBuilder.h"
#import "THTimeline.h"

@interface THAudioMixCompositionBuilder : NSObject <THCompositionBuilder>

- (id)initWithTimeline:(THTimeline *)timeline;

@end
```

```
#import "THAudioMixCompositionBuilder.h"
#import "THAudioItem.h"
#import "THVolumeAutomation.h"
#import "THAudioMixComposition.h"
#import "THFunctions.h"

@interface THAudioMixCompositionBuilder ()
@property (strong, nonatomic) THTimeline *timeline;
@property (strong, nonatomic) AVMutableComposition *composition;
@end

@implementation THAudioMixCompositionBuilder

- (id)initWithTimeline:(THTimeline *)timeline {
    self = [super init];
    if (self) {
        _timeline = timeline;
    }
    return self;
}

- (id <THComposition>)buildComposition {
    self.composition = [AVMutableComposition composition]; // 1
    [self addCompositionTrackOfType:AVMediaTypeVideo
                           withMediaItems:self.timeline.videos];
}
```

```

[self addCompositionTrackOfType:AVMediaTypeAudio
    withMediaItems:self.timeline.voiceOvers];

AVMutableCompositionTrack *musicTrack =
    [self addCompositionTrackOfType:AVMediaTypeAudio
        withMediaItems:self.timeline.musicItems];

AVAudioMix *audioMix = [self buildAudioMixWithTrack:musicTrack];           // 2

return [THAudioMixComposition compositionWithComposition:self.composition
    audioMix:audioMix];
}

- (AVMutableCompositionTrack *)addCompositionTrackOfType:(NSString *)type
    withMediaItems:(NSArray *)mediaItems {

    if (!THIsEmpty(mediaItems)) {

        CMPersistentTrackID trackID = kCMPersistentTrackID_Invalid;

        AVMutableCompositionTrack *compositionTrack =
            [self.composition addMutableTrackWithMediaType:type
                preferredTrackID:trackID];
        // Set insert cursor to 0
        CMTime cursorTime = kCMTimeZero;

        for (THMediaItem *item in mediaItems) {

            if (CMTIME_COMPARE_INLINE(item.startTimeInTimeline,
                !=,
                kCMTimeInvalid)) {
                cursorTime = item.startTimeInTimeline;
            }

            AVAssetTrack *assetTrack =
                [[item.asset tracksWithMediaType:type] firstObject];

            [compositionTrack insertTimeRange:item.timeRange
                ofTrack:assetTrack
                atTime:cursorTime
                error:nil];

            // Move cursor to next item time
            cursorTime = CMTimeAdd(cursorTime, item.timeRange.duration);
        }
    }
}

```

```
        return compositionTrack;
    }

    return nil;
}

- (AVAudioMix *)buildAudioMixWithTrack:(AVMutableCompositionTrack *)track {

    // To be implemented

    return nil;
}

@end
```

```
- (AVAudioMix *)buildAudioMixWithTrack:(AVCompositionTrack *)track {
    THAudioItem *item = [self.timeline.musicItems firstObject];           // 1
    if (item) {
        AVMutableAudioMix *audioMix = [AVMutableAudioMix audioMix];          // 2
        AVMutableAudioMixInputParameters *parameters =
            [AVMutableAudioMixInputParameters
                audioMixInputParametersWithTrack:track];
        for (THVolumeAutomation *automation in item.volumeAutomation) {      // 3
            [parameters setVolumeRampFromStartVolume:automation.startVolume
                toEndVolume:automation.endVolume
                timeRange:automation.timeRange];
        }
        audioMix.inputParameters = @[parameters];
        return audioMix;
    }
    return nil;
}
```

```
AVMutableComposition *composition = [AVMutableComposition composition];

AVMutableCompositionTrack *trackA =
    [composition addMutableTrackWithMediaType:AVMediaTypeVideo
        preferredTrackID:kCMPersistentTrackID_Invalid];

AVMutableCompositionTrack *trackB =
    [composition addMutableTrackWithMediaType:AVMediaTypeVideo
        preferredTrackID:kCMPersistentTrackID_Invalid];

NSArray *videoTracks = @[trackA, trackB];
```

```
NSArray *videoAssets = nil;// array of loaded AVAsset instances;

CMTime cursorTime = kCMTimeZero;

for (NSUInteger i = 0; i < videoAssets.count; i++) {

    NSUInteger trackIndex = i % 2;

    AVMutableCompositionTrack *currentTrack = videoTracks[trackIndex];

    AVAsset *asset = videoAssets[i];
    AVAssetTrack *assetTrack =
        [[asset tracksWithMediaType:AVMediaTypeVideo] firstObject];

    CMTimeRange timeRange = CMTimeRangeMake(kCMTimeZero, asset.duration);

    [currentTrack insertTimeRange:timeRange
                           ofTrack:assetTrack
                           atTime:cursorTime error:nil];

    cursorTime = CMTimeAdd(cursorTime, timeRange.duration);
}
```

```
NSArray *videoAssets = nil;// array of loaded AVAsset instances;

CMTime cursorTime = kCMTimeZero;
CMTime transitionDuration = CMTimeMake(2, 1);

for (NSUInteger i = 0; i < videoAssets.count; i++) {

    NSUInteger trackIndex = i % 2;

    AVMutableCompositionTrack *currentTrack = videoTracks[trackIndex] ;

    AVAsset *asset = videoAssets[i];
    AVAssetTrack *assetTrack =
        [[asset tracksWithMediaType:AVMediaTypeVideo] firstObject];

    CMTimeRange timeRange = CMTimeRangeMake(kCMTimeZero, asset.duration);

    [currentTrack insertTimeRange:timeRange
                           ofTrack:assetTrack
                           atTime:cursorTime error:nil];

    cursorTime = CMTimeAdd(cursorTime, timeRange.duration);
    cursorTime = CMTimeSubtract(cursorTime, transitionDuration);
}
```

```

NSMutableArray *videoAssets = // loaded video assets

CMTime cursorTime = kCMTimeZero;

// 2 second transition duration
CMTime transDuration = CMTimeMake(2, 1);

NSMutableArray *passThroughTimeRanges = [NSMutableArray array];
NSMutableArray *transitionTimeRanges = [NSMutableArray array];

NSUInteger videoCount = [videoAssets count];

for (NSUInteger i = 0; i < videoCount; i++) {

    AVAsset *asset = videoAssets[i];

    CMTimeRange timeRange = CMTimeRangeMake(cursorTime, asset.duration);

    if (i > 0) {
        timeRange.start = CMTimeAdd(timeRange.start, transDuration);
        timeRange.duration = CMTimeSubtract(timeRange.duration, transDuration);
    }

    if (i + 1 < videoCount) {
        timeRange.duration = CMTimeSubtract(timeRange.duration, transDuration);
    }

    [passThroughTimeRanges addObject:[NSValue valueWithCMTimeRange:timeRange]];

    cursorTime = CMTimeAdd(cursorTime, asset.duration);
    cursorTime = CMTimeSubtract(cursorTime, transDuration);

    if (i + 1 < videoCount) {
        timeRange = CMTimeRangeMake(cursorTime, transDuration);
        NSValue *timeRangeValue = [NSValue valueWithCMTimeRange:timeRange];
        [transitionTimeRanges addObject:timeRangeValue];
    }
}

```

```
NSMutableArray *compositionInstructions = [NSMutableArray array];

// Look up all of the video tracks in the composition
NSArray *tracks = [composition tracksWithMediaType:AVMediaTypeVideo];

for (NSUInteger i = 0; i < passThroughTimeRanges.count; i++) { // 1

    // Calculate the trackIndex to operate upon: 0, 1, 0, 1, etc.
    NSUInteger trackIndex = i % 2;

    AVMutableCompositionTrack *currentTrack = tracks[trackIndex];

    AVMutableVideoCompositionInstruction *instruction = // 2
        [AVMutableVideoCompositionInstruction videoCompositionInstruction];

    instruction.timeRange = [passThroughTimeRanges[i] CMTimeRangeValue];

    AVMutableVideoCompositionLayerInstruction *layerInstruction = // 3
        [AVMutableVideoCompositionLayerInstruction
            videoCompositionLayerInstructionWithAssetTrack:currentTrack];

    instruction.layerInstructions = @[layerInstruction];

    [compositionInstructions addObject:instruction];

    if (i < transitionTimeRanges.count) {
```

```
AVCompositionTrack *foregroundTrack = tracks[trackIndex]; // 4
AVCompositionTrack *backgroundTrack = tracks[1 - trackIndex];

AVMutableVideoCompositionInstruction *instruction = // 5
    [AVMutableVideoCompositionInstruction videoCompositionInstruction];

CMTTimeRange timeRange = [transitionTimeRanges[i] CMTTimeRangeValue];
instruction.timeRange = timeRange;

AVMutableVideoCompositionLayerInstruction *fromLayerInstruction = // 6
    [AVMutableVideoCompositionLayerInstruction
        videoCompositionLayerInstructionWithAssetTrack:foregroundTrack];

AVMutableVideoCompositionLayerInstruction *toLayerInstruction =
    [AVMutableVideoCompositionLayerInstruction
        videoCompositionLayerInstructionWithAssetTrack:backgroundTrack];

instruction.layerInstructions = @[fromLayerInstruction, // 7
                                toLayerInstruction];

[compositionInstructions addObject:instruction];
}

}
```

```
AVMutableVideoComposition *videoComposition =
    [AVMutableVideoComposition videoComposition];
videoComposition.instructions = compositionInstructions;
videoComposition.renderSize = CGSizeMake(1280.0f, 720.0f);
videoComposition.frameDuration = CMTimeMake(1, 30);
videoComposition.renderScale = 1.0f;
```

```
#import "THComposition.h"

@interface THTransitionComposition : NSObject <THComposition>

@property (strong, nonatomic, readonly) AVComposition *composition;
@property (strong, nonatomic, readonly) AVVideoComposition *videoComposition;
@property (strong, nonatomic, readonly) AVAudioMix *audioMix;

- (id)initWithComposition:(AVComposition *)composition
                  videoComposition:(AVVideoComposition *)videoComposition
                        audioMix:(AVAudioMix *)audioMix;
@end
```

```
#import "THTransitionComposition.h"

@implementation THTransitionComposition

- (id)initWithComposition:(AVComposition *)composition
    videoComposition:(AVVideoComposition *)videoComposition
        audioMix:(AVAudioMix *)audioMix {
    self = [super init];
    if (self) {
        _composition = composition;
        _videoComposition = videoComposition;
        _audioMix = audioMix;
    }
    return self;
}

- (AVPlayerItem *)makePlayable { // 1
    AVPlayerItem *playerItem =
    [AVPlayerItem playerItemWithAsset:[self.composition copy]];

    playerItem.audioMix = self.audioMix;
    playerItem.videoComposition = self.videoComposition;

    return playerItem;
}

- (AVAssetExportSession *)makeExportable { // 2
    NSString *preset = AVAssetExportPresetHighestQuality;
    AVAssetExportSession *session =
    [AVAssetExportSession exportSessionWithAsset:[self.composition copy]
        presetName:preset];
    session.audioMix = self.audioMix;
    session.videoComposition = self.videoComposition;

    return session;
}

@end
```

```
#import "THCompositionBuilder.h"
#import "THTimeline.h"

@interface THTransitionCompositionBuilder : NSObject <THCompositionBuilder>

- (id)initWithTimeline:(THTimeline *)timeline;

@end
```

```
#import "THTransitionCompositionBuilder.h"
#import "THVideoItem.h"
#import "THAudioItem.h"
#import "THVolumeAutomation.h"
#import "THTransitionComposition.h"
#import "THTransitionInstructions.h"
#import "THFunctions.h"

@interface THTransitionCompositionBuilder ()
@property (strong, nonatomic) THTimeline *timeline;
@property (strong, nonatomic) AVMutableComposition *composition;
@property (weak, nonatomic) AVMutableCompositionTrack *musicTrack;
@end

@implementation THTransitionCompositionBuilder

- (id)initWithTimeline:(THTimeline *)timeline {
    self = [super init];
    if (self) {
        _timeline = timeline;
    }
    return self;
}

- (id <THComposition>)buildComposition {
    self.composition = [AVMutableComposition composition];
    [self buildCompositionTracks];
}
```

```
AVVideoComposition *videoComposition = [self buildVideoComposition];

AVAudioMix *audioMix = [self buildAudioMix];

return [[THTransitionComposition alloc] initWithComposition:self.composition
                                                 videoComposition:videoComposition
                                                 audioMix:audioMix];
}

- (void)buildCompositionTracks {
    // To be implemented
}

- (AVVideoComposition *)buildVideoComposition {
    // To be implemented
    return nil;
}

@end
```

```
- (void)buildCompositionTracks {

    CMPersistentTrackID trackID = kCMPersistentTrackID_Invalid;
    AVMutableCompositionTrack *compositionTrackA =
        [self.composition addMutableTrackWithMediaType:AVMediaTypeVideo
            preferredTrackID:trackID]; // 1

    AVMutableCompositionTrack *compositionTrackB =
        [self.composition addMutableTrackWithMediaType:AVMediaTypeVideo
            preferredTrackID:trackID];

    NSArray *videoTracks = @[compositionTrackA, compositionTrackB];

    CMTime cursorTime = kCMTimeZero;
    CMTime transitionDuration = kCMTimeZero;

    if (!THIsEmpty(self.timeline.transitions)) { // 2
        // 1 second transition duration
        transitionDuration = THDefaultTransitionDuration;
    }

    NSArray *videos = self.timeline.videos;

    for (NSUInteger i = 0; i < videos.count; i++) {
        NSUInteger trackIndex = i % 2; // 3

        THVideoItem *item = videos[i];
        AVMutableCompositionTrack *currentTrack = videoTracks[trackIndex];
    }
}
```

```
AVAssetTrack *assetTrack =
    [[item.asset tracksWithMediaType:AVMediaTypeVideo] firstObject];

[currentTrack insertTimeRange:item.timeRange
    ofTrack:assetTrack
    atTime:cursorTime error:nil];

// Overlap clips by transition duration // 4
cursorTime = CMTimeAdd(cursorTime, item.timeRange.duration);
cursorTime = CMTimeSubtract(cursorTime, transitionDuration);
}

// Add voice overs // 5
[self addCompositionTrackOfType:AVMediaTypeAudio
    withMediaItems:self.timeline.voiceOvers];

// Add music track
NSArray *musicItems = self.timeline.musicItems;
self.musicTrack = [self addCompositionTrackOfType:AVMediaTypeAudio
    withMediaItems:musicItems];
}
```

```
- (AVVideoComposition *)buildVideoComposition {

    AVVideoComposition *videoComposition = // 1
    [AVMutableVideoComposition
        videoCompositionWithPropertiesOfAsset:self.composition];

    NSArray *transitionInstructions = // 2
    [self transitionInstructionsInVideoComposition:videoComposition];

    for (THTransitionInstructions *instructions in transitionInstructions) {

        CMTimeRange timeRange = // 3
        instructions.compositionInstruction.timeRange;

        AVMutableVideoCompositionLayerInstruction *fromLayer =
        instructions.fromLayerInstruction;

        AVMutableVideoCompositionLayerInstruction *toLayer =
        instructions.toLayerInstruction;

        THVideoTransitionType type = instructions.transition.type;
    }
}
```

```
// Apply Video Transition Effects // 4
if (type == THVideoTransitionTypeDissolve) {
    // Listing 11.7
}

else if (type == THVideoTransitionTypePush) {
    // Listing 11.8
}

else (type == THVideoTransitionTypeWipe) {
    // Listing 11.9
}

instructions.compositionInstruction.layerInstructions = @[fromLayer, // 5
                                                       toLayer];
}

return videoComposition;
}

- (NSArray *)transitionInstructionsInVideoComposition:(AVVideoComposition *)vc {
    // To be implemented

    return nil;
}
```

```
- (NSArray *)transitionInstructionsInVideoComposition:(AVVideoComposition *)vc {

    NSMutableArray *transitionInstructions = [NSMutableArray array];

    int layerInstructionIndex = 1;

    NSArray *compositionInstructions = vc.instructions; // 1

    for (AVMutableVideoCompositionInstruction *vci in compositionInstructions) {

        if (vci.layerInstructions.count == 2) { // 2

            THTransitionInstructions *instructions =
                [[THTransitionInstructions alloc] init];

            instructions.compositionInstruction = vci;

            instructions.fromLayerInstruction =
                vci.layerInstructions[1 - layerInstructionIndex]; // 3

            instructions.toLayerInstruction =
                vci.layerInstructions[layerInstructionIndex];

            [transitionInstructions addObject:instructions];
        }

        layerInstructionIndex = layerInstructionIndex == 1 ? 0 : 1;
    }
}

NSArray *transitions = self.timeline.transitions;

// Transitions are disabled
if (THIsEmpty(transitions)) { // 4
    return transitionInstructions;
}

NSAssert(transitionInstructions.count == transitions.count,
         @"Instruction count and transition count do not match.");

for (NSUInteger i = 0; i < transitionInstructions.count; i++) { // 5
    THTransitionInstructions *tis = transitionInstructions[i];
    tis.transition = self.timeline.transitions[i];
}

return transitionInstructions;
}
```

```
if (type == THVideoTransitionTypeDissolve) {  
    [fromLayer setOpacityRampFromStartOpacity:1.0  
                 toEndOpacity:0.0  
                 timeRange:timeRange] ;  
}
```

```
else (type == THVideoTransitionTypeWipe) {  
  
    CGFloat videoWidth = videoComposition.renderSize.width;  
    CGFloat videoHeight = videoComposition.renderSize.height;  
  
    CGRect startRect = CGRectMake(0.0f, 0.0f, videoWidth, videoHeight);  
    CGRect endRect = CGRectMake(0.0f, videoHeight, videoWidth, 0.0f);  
  
    [fromLayer setCropRectangleRampFromStartCropRectangle:startRect  
        toEndCropRectangle:endRect  
        timeRange:timeRange] ;  
}
```

```
CALayer *parentLayer = // parent layer

UIImage *image = [UIImage imageNamed:@"lavf_cover"] ;

CALayer *imageLayer = [CALayer layer] ;
// Set the layer contents to the book cover image
imageLayer.contents = (id)image.CGImage;
imageLayer.contentsScale = [UIScreen mainScreen].scale;

// Size and position the layer
CGFloat midX = CGRectGetMidX(parentLayer.bounds) ;
CGFloat midY = CGRectGetMidY(parentLayer.bounds) ;

imageLayer.bounds = CGRectMake(0, 0, image.size.width, image.size.height) ;
imageLayer.position = CGPointMake(midX, midY) ;

// Add the image layer as a sublayer of the parent layer
[parentLayer addSublayer:imageLayer] ;

// Basic animation to rotate around z-axis
CABasicAnimation *rotationAnimation =
[CABasicAnimation animationWithKeyPath:@"transform.rotation.z"] ;

// Rotate 360 degrees over a three-second duration, repeat indefinitely
rotationAnimation.toValue = @(2 * M_PI) ;
rotationAnimation.duration = 3.0f;
rotationAnimation.repeatCount = HUGE_VALF;

// Add and execute animation on the image layer
[imageLayer addAnimation:rotationAnimation forKey:@"rotateAnimation"] ;
```

```
#import "THTimelineItem.h"
#import <QuartzCore/QuartzCore.h>

@interface THTitleItem : THTimelineItem

+ (instancetype)titleItemWithText:(NSString *)text image:(UIImage *)image;
- (instancetype)initWithText:(NSString *)text image:(UIImage *)image;

@property (copy, nonatomic) NSString *identifier;
@property (nonatomic) BOOL animateImage;
@property (nonatomic) BOOL useLargeFont;

- (CALayer *)buildLayer;

@end
```

```
#import "THTitleItem.h"
#import "THConstants.h"

@interface THTitleItem ()
@property (copy, nonatomic) NSString *text;
@property (strong, nonatomic) UIImage *image;
@property (nonatomic) CGRect bounds;
@end

@implementation THTitleItem

+ (instancetype)titleItemWithText:(NSString *)text image:(UIImage *)image {
    return [[self alloc] initWithText:text image:image];
}

- (instancetype)initWithText:(NSString *)text image:(UIImage *)image {
    self = [super init];
    if (self) {
        _text = [text copy];
        _image = image;
        _bounds = TH720pVideoRect; // 1
    }
    return self;
}

- (CALayer *)buildLayer {
    // --- Build Layers
```

```
CALayer *parentLayer = [CALayer layer]; // 2
parentLayer.frame = self.bounds;
parentLayer.opacity = 0.0f;

CALayer *imageLayer = [self makeImageLayer];
[parentLayer addSublayer:imageLayer];

CALayer *textLayer = [self makeTextLayer];
[parentLayer addSublayer:textLayer];

return parentLayer;
}

- (CALayer *)makeImageLayer { // 3

    CGSize imageSize = self.image.size;

    CALayer *layer = [CALayer layer];
    layer.contents = (id) self.image.CGImage;
    layer.allowsEdgeAntialiasing = YES;
    layer.bounds = CGRectMake(0.0f, 0.0f, imageSize.width, imageSize.height);
    layer.position = CGPointMake(CGRectGetMidX(self.bounds) - 20.0f, 270.0f);

    return layer;
}
```

```
- (CALayer *)makeTextLayer { // 4

    CGFloat fontSize = self.useLargeFont ? 64.0f : 54.0f;
    UIFont *font = [UIFont fontWithName:@"GillSans-Bold" size:fontSize];

    NSDictionary *attrs =
        @{@"NSFontAttributeName" : font,
         NSForegroundColorAttributeName : (id) [UIColor whiteColor].CGColor};

    NSAttributedString *string =
        [[NSAttributedString alloc] initWithString:self.text attributes:attrs];

    CGSize textSize = [self.text sizeWithAttributes:attrs];

    CATextLayer *layer = [CATextLayer layer];
    layer.string = string;
    layer.bounds = CGRectMake(0.0f, 0.0f, textSize.width, textSize.height);
    layer.position = CGPointMake(CGRectGetMidX(self.bounds), 470.0f);
    layer.backgroundColor = [UIColor clearColor].CGColor;

    return layer;
}

@end
```

```
@implementation THTitleItem

...
- (CALayer *)buildLayer {
    // --- Build Layers
    CALayer *parentLayer = [CALayer layer];
    parentLayer.frame = self.bounds;
    parentLayer.opacity = 0.0f;

    CALayer *imageLayer = [self makeImageLayer];
    [parentLayer addSublayer:imageLayer];

    CALayer *textLayer = [self makeTextLayer];
    [parentLayer addSublayer:textLayer];

    // --- Build and Attach Animations
    CAAcceleration *fadeInFadeOutAnimation = [self makeFadeInFadeOutAnimation];
    [parentLayer addAnimation:fadeInFadeOutAnimation forKey:nil];           // 1

    return parentLayer;
}

...
- (CAAnimation *)makeFadeInFadeOutAnimation {
    CAKeyframeAnimation *animation =
        [CAKeyframeAnimation animationWithKeyPath:@"opacity"];

    animation.values = @[@0.0f, @1.0, @1.0f, @0.0f];                      // 2
    animation.keyTimes = @[@0.0f, @0.20f, @0.80f, @1.0f];

    animation.beginTime = CMTimeGetSeconds(self.startTimeInTimeline);        // 3
    animation.duration = CMTimeGetSeconds(self.timeRange.duration);

    animation.removedOnCompletion = NO;                                       // 4

    return animation;
}

@end
```

```
- (CALayer *)buildLayer {

    // --- Build Layers

    CALayer *parentLayer = [CALayer layer];
    parentLayer.frame = self.bounds;
    parentLayer.opacity = 0.0f;

    CALayer *imageLayer = [self makeImageLayer];
    [parentLayer addSublayer:imageLayer];

    CALayer *textLayer = [self makeTextLayer];
    [parentLayer addSublayer:textLayer];

    // --- Build and Attach Animations

    CAAnimation *fadeInFadeOutAnimation = [self makeFadeInFadeOutAnimation];
    [parentLayer addAnimation:fadeInFadeOutAnimation forKey:nil];

    if (self.animateImage) {

        parentLayer.sublayerTransform = THMakePerspectiveTransform(1000); // 1

        CAAnimation *spinAnimation = [self make3DSpinAnimation];
```

```

NSTimeInterval offset = // 2
    spinAnimation.beginTime + spinAnimation.duration - 0.5f;

CAAnimation *popAnimation =
    [self makePopAnimationWithTimingOffset:offset];

[imageLayer addAnimation:spinAnimation forKey:nil]; // 3
[imageLayer addAnimation:popAnimation forKey:nil];

}

return parentLayer;
}

...

static CATransform3D THMakePerspectiveTransform(CGFloat eyePosition) {
    CATransform3D transform = CATransform3DIdentity;
    transform.m34 = -1.0 / eyePosition;
    return transform;
}

- (CAAnimation *)make3DSpinAnimation {

    // To be implemented

    return nil;
}

- (CAAnimation *)makePopAnimationWithTimingOffset:(NSTimeInterval)offset {

    // To be implemented

    return nil;
}

@end

```

```
- (CAAnimation *)make3DSpinAnimation {

    CABasicAnimation *animation = // 1
        [CABasicAnimation animationWithKeyPath:@"transform.rotation.y"];

    animation.toValue = @((4 * M_PI) * -1); // 2

    animation.beginTime = CMTimeGetSeconds(self.startTimeInTimeline) + 0.2; // 3
    animation.duration = CMTimeGetSeconds(self.timeRange.duration) * 0.4;

    animation.removedOnCompletion = NO;

    animation.timingFunction = // 4
        [CAMediaTimingFunction
            functionWithName:kCAMediaTimingFunctionEaseInEaseOut];

    return animation;
}

- (CAAnimation *)makePopAnimationWithTimingOffset:(NSTimeInterval)offset {

    CABasicAnimation *animation = // 5
        [CABasicAnimation animationWithKeyPath:@"transform.scale"];

    animation.toValue = @1.3f; // 6

    animation.beginTime = offset; // 7
    animation.duration = 0.35f;

    animation.autoreverses = YES; // 8

    animation.removedOnCompletion = NO;

    animation.timingFunction =
        [CAMediaTimingFunction
            functionWithName:kCAMediaTimingFunctionEaseInEaseOut];

    return animation;
}
```

```
@interface THOverlayComposition : NSObject <THComposition>

@property (strong, nonatomic, readonly) AVComposition *composition;
@property (strong, nonatomic, readonly) AVVideoComposition *videoComposition;
@property (strong, nonatomic, readonly) AVAudioMix *audioMix;
@property (strong, nonatomic, readonly) CALayer *titleLayer;

- (id)initWithComposition:(AVComposition *)composition
    videoComposition:(AVVideoComposition *)videoComposition
        audioMix:(AVAudioMix *)audioMix
        titleLayer:(CALayer *)titleLayer;
@end
```

```
@implementation THOverlayCompositionBuilder

...
- (id <THComposition>)buildComposition {
    self.composition = [AVMutableComposition composition];
    [self buildCompositionTracks];
    AVVideoComposition *videoComposition = [self buildVideoComposition];
    return [[THOverlayComposition alloc] // 1
            initWithComposition:self.composition
            videoComposition:videoComposition
            audioMix:[self buildAudioMix]
            titleLayer:[self buildTitleLayer]];
}
- (CALayer *)buildTitleLayer {
    if (!THIsEmpty(self.timeline.titles)) {
        CALayer *titleLayer = [CALayer layer]; // 2
        titleLayer.bounds = TH720pVideoRect;
        titleLayer.position = CGPointMake(CGRectGetMidX(TH720pVideoRect),
                                         CGRectGetMidY(TH720pVideoRect));
        for (THTitleItem *compositionLayer in self.timeline.titles) { // 3
            [titleLayer addSublayer:[compositionLayer buildLayer]];
        }
        return titleLayer;
    }
    return nil;
}
...
@end
```

```
- (AVPlayerItem *)makePlayable {

    AVPlayerItem *playerItem =
        [AVPlayerItem playerItemWithAsset:[self.composition copy]];

    playerItem.videoComposition = self.videoComposition;
    playerItem.audioMix = self.audioMix;

    if (self.titleLayer) { // 1
        AVSynchronizedLayer *syncLayer =
            [AVSynchronizedLayer synchronizedLayerWithPlayerItem:playerItem];

        [syncLayer addSublayer:self.titleLayer];

        // WARNING: The the 'titleLayer' property is NOT part of AV Foundation
        // Provided by AVPlayerItem+THAdditions category.
        playerItem.syncLayer = syncLayer; // 2
    }

    return playerItem;
}
```

```
- (AVAssetExportSession *)makeExportable {

    if (self.titleLayer) {

        CALayer *animationLayer = [CALayer layer]; // 1
        animationLayer.frame = TH720pVideoRect;

        CALayer *videoLayer = [CALayer layer];
        videoLayer.frame = TH720pVideoRect;

        [animationLayer addSublayer:videoLayer];
        [animationLayer addSublayer:self.titleLayer];

        animationLayer.geometryFlipped = YES; // 2

        AVVideoCompositionCoreAnimationTool *animationTool =
            ➔ [AVVideoCompositionCoreAnimationTool
            ➔ videoCompositionCoreAnimationToolWithPostProcessingAsVideoLayer:videoLayer
            ➔ inLayer:animationLayer];

        AVMutableVideoComposition *mvc =
            (AVMutableVideoComposition *)self.videoComposition;

        mvc.animationTool = animationTool; // 4
    }

    NSString *presetName = AVAssetExportPresetHighestQuality;
    AVAssetExportSession *session =
        [[AVAssetExportSession alloc] initWithAsset:[self.composition copy]
            presetName:presetName];
    session.audioMix = self.audioMix;
    session.videoComposition = self.videoComposition;

    return session;
}
```
