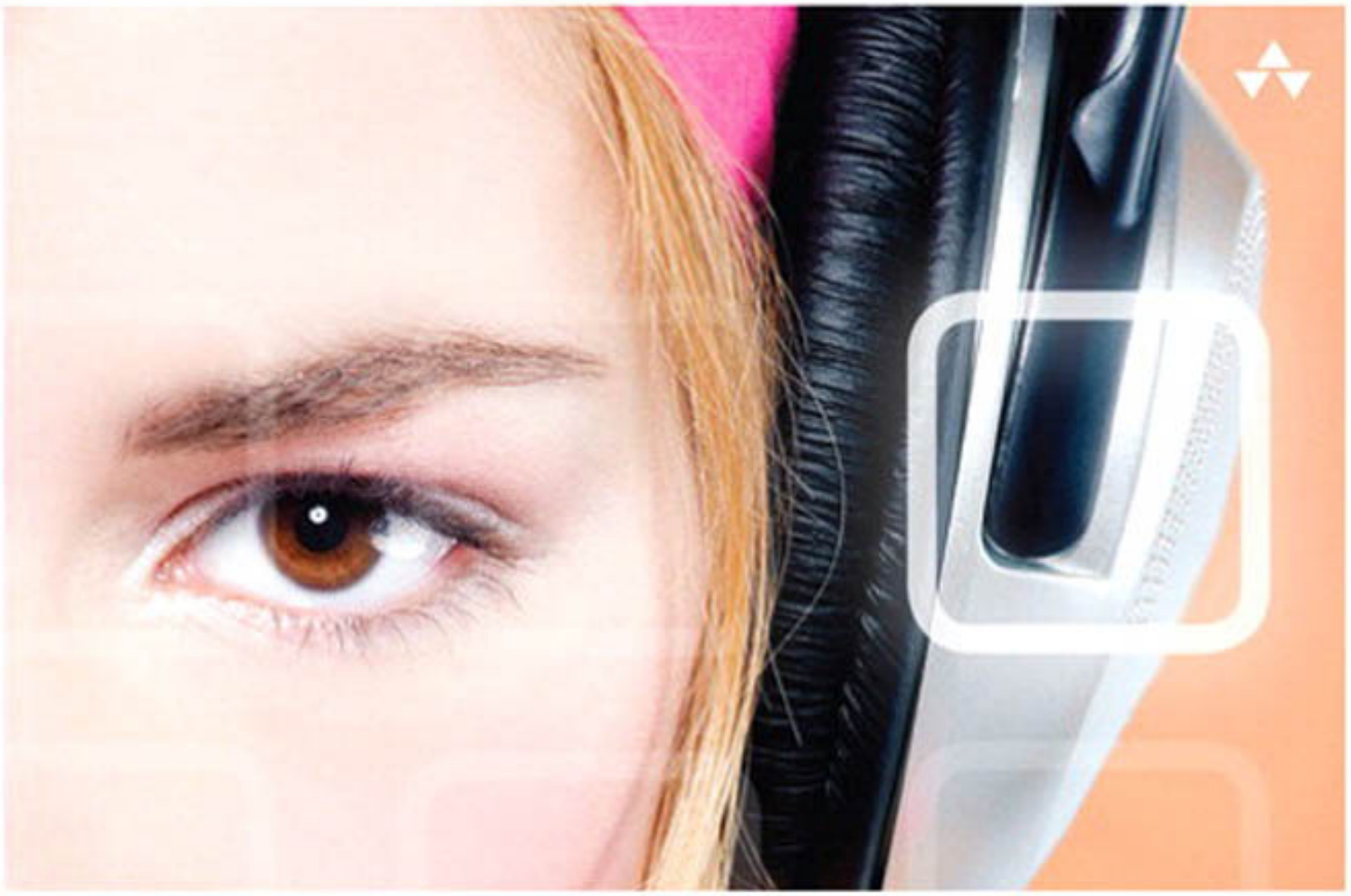

LEARNING Core Audio

A Hands-On Guide to Audio Programming for Mac and iOS

CHRIS ADAMSON
KEVIN AVILA

Learning Core Audio

Addison-Wesley Learning Series

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.

Learning Core Audio

A Hands-On Guide to Audio Programming for Mac and iOS

Chris Adamson
Kevin Avila

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

Library of Congress Cataloging-in-Publication Data

Adamson, Chris, 1967-
Learning Core audio : a hands-on guide to audio programming for Mac and iOS / Chris Adamson, Kevin Avila.
p. cm.
ISBN 978-0-321-63684-3 (pbk. : alk. paper) — ISBN 0-321-63684-8 (pbk. : alk. paper)
1. Computer sound processing—Computer programs. 2. Core audio. 3. Apple computer—Programming. I. Avila, Kevin, 1980- II. Title.
TK7881.4.A244 2012
006.4'5—dc23

2012000862

ISBN-13: 978-0-32-163684-3
ISBN-10: 0-32-163684-8

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

Second printing, June 2012

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Senior Project Editor
Lori Lyons

Copy Editor
Krista Hansing
Editorial Services, Inc.

Senior Indexer
Cheryl Lenser

Proofreader
Kathy Ruiz

Technical Reviewers
Mark Dalrymple
Mark Granoff
Michael James
Chris Liscio
Robert Strogan
Alex Wiltschko

Publishing Coordinator
Olivia Basegio

Multimedia Developer
Dan Scherf

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

Contents

This page intentionally left blank

Acknowledgments

From Chris Adamson

This book wouldn't exist without Kevin Avila and Mike Lee, who found a publisher who not only wasn't scared off by the thought of a difficult niche Mac and iOS title, but actually relished the challenge of bringing this beast to market. They knew there was a crowd out there that has been aching for years to get Core Audio presented in a practical form that lets normal programmers draw out its ferocious power. Behind the scenes, Chuck Toporek championed this book, pulled me in when it got stuck, and saw it through to the finish. More than anyone else, he's the one to thank for finally getting a Core Audio book published.

We wouldn't have been able to get it all done without the generous support of the Core Audio developer community, particularly the membership of the `coreaudio-api` mailing list. Core Audio founder William Stewart and Apple's Doug Wyatt have long been generous with their time and attention to questions posted to the list and got us unstuck on a number of occasions.

We're also grateful to our many tech reviewers and readers of the "Rough Cuts" edition who reported errors and provided feedback as this book worked through its long road to completion.

At home, thanks to my wife, Kelly, and our kids, Keagan and Quinn, for cutting me enough slack to get this thing done and not completely freaking out when the example code went wrong and horrible buzzes blasted forth from Dad's office in the basement.

Obligatory end-of-book tune check: This time it was We Are The City, … And You Will Know Us by the Trail of Dead, Daft Punk, Dr. Dog, Fun, and (fittingly) Hatsune Miku.[1]

From Kevin Avila

I would like to acknowledge the Big Bang, gravity, and the eventual copulation between my parents for making this possible.

Chuck Toporek (@chuckdude), Chris Adamson (@invalidname), Mike Lee (@bmf): There truly are no words that express my gratitude for all the blood, sweat, and grammar you've contributed, not only to this book, but to the entire developer community. Thank you.

[1] Find up-to-date listening stats at www.last.fm/user/invalidname.

Bill Stewart, Jeff Moore, Doug Wyatt, Michael Hopkins, Bob Aron, James McCartney, Mehul Trivedi, Cynthia Maxwell, Torrey Walker, Nick Thompson, Matthew Mora, Brad Ford, Murray Jason, and Edward Agabeg: Thanks for sharing with me your passion and knowledge of audio.

Special thanks to David Avila, Daniel Kaufman, Andre LaBranche, Quentin Carnicelli, Ed Wynne, and Steve Jobs.

What's on my iPod: AC/DC, Rush, Beach Boys, Sublime, Primus, KRS-One, Beastie Boys, Mac Dre, Vokab Kompany, and the insanely great George Carlin.

About the Authors

Chris Adamson is an independent writer, editor, and developer who lives in Grand Rapids, Michigan. Now focusing on iOS and Mac development, he is the coauthor of *iOS SDK Development* (Pragmatic Programmers, 2012). He is also the author of *QuickTime for Java: A Developer's Notebook* (O'Reilly Media, 2005) and coauthor of *Swing Hacks* (O'Reilly Media, 2005). He was formerly the editor of *java.net* and *ONJava.com*. He consults and publishes through his corporate identity, Subsequently and Furthermore, Inc., with a focus on user-facing and digital media development for Mac and iOS. He blogs on digital media software development at www.subfurther.com/blog. In a previous career, he was a writer/associate producer at *CNN Headline News,* and over the years, he has managed to own 11 1/2 Macs.

Kevin Avila (a.k.a. dogbert) is a smooth blend of carbon compounds, oxygen, hydrogen, and nitrogen, with some impurities for added flavor. Additionally, he has more than 15 years' experience developing for the Mac and, since its release, the iPhone. Kevin has been involved in every corner of the audio market, from being an engineer at Apple to configuring professional recording studios. He currently is a code mercenary for various clients while he sits in his underwear at home, sipping coffee.

We'd Like to Hear from You

You can visit our website and register this book at:

www.informit.com/title/9780321636843

Be sure to visit the book's website for convenient access to any updates, to download the book's sample code, or for errata that might be available for this book.

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

When you write, please be sure to include this book's title and the name of the author, as well as your name, phone, and/or e-mail address. I will carefully review your comments and share them with the author and others who have worked on this book.

E-mail: trina.macdonald@pearson.com

Mail: Trina MacDonald
Senior Acquisitions Editor, Addison-Wesley
Pearson Education, Inc.
1249 8th Street
Berkeley, CA 94710 USA

For more information about our books or conferences, see our website at:

www.informit.com

Foreword

Reflect for a minute on your craft. Think of those in ages past who shared the same experiences. Think of the painters who drove themselves mad trying to gather the forces within to produce something of meaning. Think of the industrialists, who believed they were standing at the dawn of a new age, one that they themselves were capable of building.

Think of the ancient acolytes of magic, seeking to unlock the power in arcane knowledge. Then think of that moment when, having learned street magic tricks such as flow control and data structures, you finally gained access to the API libraries. Think of that sorcerer's apprentice staring glassy-eyed at the universe of possibilities in a room of musty books.

It's one of those key moments in any programmer's career, cresting that foothill only to see the mountain beyond. It is the daunting realization that programming is a lifelong journey of learning. Many would-be magicians simply turn around and head back out the door, leaving that world behind to pursue a more normal life of sane pursuits.

That you have found your way here suggests you are part of the other group, a select few blessed with some genetic predisposition to solving hard problems. They are the ones who cross that threshold and enter that world, losing themselves to learning new spells and exploring new kinds of magic.

For what is programming but magic? Wielding secret words to command powerful forces you just barely understand, calling forth the spirits of bygone spell casters to ease your burdens, simplify your workflows, and grant you the ability to surprise and delight the masses.

As you pore over each tome, practicing new forms of magic, you start combining them with the things you learned before creating new spells, thus unlocking new possibilities. Everything you learn leads to new tricks to learn, new roads forking into other new roads, until one day you run into a dead end.

Many of the ancient texts refer to such dark arts as audio programming but do not deal with them directly. As vital as Core Audio is, there seem to be no books on the subject, no scrolls of spells or sample code to practice. There are plenty of use cases for audio programming, but as black magic, the only materials you can find to read about it are terse and confusing.

Chris Adamson tracked down a practitioner of the audio arts named Kevin Avila, a graying wizard well versed in C. Through a combination of bribery and honest inquiry, he established himself as a protégé. The rabbit hole he entered goes on forever, and as his

ears led him through its many dark twists and turns, he learned a new language to describe sound—and, with it, a new way of looking at the universe.

An eternity later, he himself a graying wizard, he thought back on that library to the missing volumes and realized it was his destiny to shed light on the dark art of Core Audio. It is the definition of mastery that we must teach what we have learned. This is the truth that fuels the cycle of master and protégé. This is the engine that drives generations forward, each propelling the next further ahead as we move toward that grand ineffable vanishing point we call the future.

As with all rites of passage, it was a herculean task, requiring a whole new sets of skills and a different type of discipline. We must tear apart our knowledge, and ourselves, to find not just truth, but the beauty that underlies the truth and allows it to resonate across the ages and to be understood.

All such that at some unknowable time in the future, where once there was a dead end and a blank space between Core Animation and Core Data, some young acolyte might find wisdom and guidance. They might combine this new knowledge with what they already know so that, when they find their own dead end and their own dark arts, they, too, will be ready.

That moment, dear reader, is now. That acolyte is you, and the grimoire that you hold in your hand has all the wisdom and more than enough spells to take your magic to the next level. This book is your key to wielding unspeakable power, the power of sound and nature, the power of Core Audio.

Does all that seem a bit much for a book about audio programming? Rest assured that, if anything, I have undersold it. Sound is an incredibly powerful force that affects the human brain in ways we only barely understand. Consider the impact of music on your life. Now consider that all of music is maybe 10% of the story of sound.

The power of audio programming goes so far beyond anything you can experience with your ears. Swiping a credit card used to require an expensive machine. Now you can do the same trick with a cheap plastic dongle plugged into the headphone jack of your iPhone. You don't have to make music to make magic with sound.

With this book, you dig into your first Core Audio code in Chapter 1, "Overview of Core Audio," even as you are learning what exactly Core Audio is and when you should (and should not) attempt to use its power.

Core Audio, like all black arts, has roots in the inherent properties of nature. Chapter 2, "The Story of Sound," takes to heart the story of sound, not as ineffable natural phenomena, but as simple science. You'll learn the language and techniques of converting vibrating air molecules into the mathematical language of computers, and vice versa.

You'll also learn the human language of audio and the real meanings of technical terms you've heard, and perhaps even used, for years: sample rate, frame rate, buffer, and compression. You'll see these ideas carried through Chapter 3, "Audio Processing with Core Audio," as you peel back the wrapper on audio formats and learn about the canonical formats Core Audio uses internally.

When you know the basics of Core Audio, you'll want to apply your skills by learning the parlor tricks of recording and playback with Chapters 4, "Recording," and 5, "Playback," using the high-level Audio Queue architecture.

Of course, "high-level" can be a misnomer, especially if you're coming from an object-oriented background such as Cocoa. Setting aside the comforting warmth of Objective-C to take the reins of C can certainly be scary, but with a little understanding, you'll come to see how much like Cocoa a C framework can be, as familiar friends, like key-value pairs, emerge in unfamiliar clothes.

When you understand Audio Queues, you'll be a master of audio formats—almost. First you must complete your quest by learning to convert between formats and come to understand the relevance of canonical formats.

Then it's time to say goodbye to high-level shores as you strap on your diving suit and descend into the depths of Core Audio, the modular Audio Units that implement the magic. Chapters 7, "Audio Units: Generators, Effects, and Rendering," and 8, "Audio Units: Input and Mixing," will make or break you as an audio programmer, for here you can craft end-to-end sonic solutions that are not possible "the easy way."

Once time is your plaything, it's time to tackle space. In Chapter 9, "Positional Sound," you enter another dimension as you learn to change sounds by positioning audio in space using OpenAL, the 3D audio framework.

Core Audio has its roots in the Mac but has evolved with Apple's fortunes. In Chapter 10, "Core Audio on iOS," you focus on iOS and the challenges and changes brought by the post-PC world of ultraportable hardware running ultra-efficient software.

Mobile hardware is not the only way to take audio beyond the computer. In Chapter 11, "Core MIDI," you gain the means to connect the computer to musical instruments and other hardware using Core Audio's implementation of the industry-standard Musical Instrument Digital Interface, Core MIDI.

With that, you'll be at the end of your quest, but your journey will have just begun. In Chapter 12, "Coda," you look to the future, to the once inexplicable advanced concepts you are now equipped to tackle, such as digital signal processing and sampling.

If you want to be a master of the arcane arts, you have a long road ahead of you. There's no sense sugarcoating it: This is going to be hard. But don't worry—you're in good hands. Your authors have used plain language and plenty of sample code to banish the demons and show you the way to the underlying logic that will make these concepts yours.

Core Audio is the most powerful system for audio programming man has yet to create, but its power has largely remained out of the hands of most app makers and locked in the brains of audio nerds like Kevin. Chris has done what nobody else has managed to do and may never manage to do again: Explain Core Audio in a way other people can understand.

This book has been years in the making, and it took an incredible amount of work and the best tech editor in the industry, the legendary Chuck Toporek, and his talented colleagues at Pearson to finally bring it into existence. The people into whose waiting

hands this enchanted volume has been given will be the people who deliver the coming wave of incredible audio apps.

Imagine the possibilities of connecting to people in new ways with the magic of sound. That incredible future is yours to invent. It is the dawning of the age of magic in computing, and you are a magician. Mastering the Core Audio frameworks will change the way you think about the world.

Mike Lee, Amsterdam

Introduction

Macs are great media computers, and the iPhone is the best iPod ever made—but how did they get that way? How did some of the first iOS applications turn the iPhone into a virtual instrument, yet developers on other mobile platforms remain happy enough to just to reskin another simple MP3 player? Why is the Mac the choice of so many digital media professionals, and what secret makes applications such as Bias Peak, Logic, and Soundtrack Pro possible?

Core Audio, that's what.

Core Audio is the low-level API that Apple provides for working with digital audio on Mac OS X and iOS. It provides APIs for simultaneously processing many streams of multichannel digital audio and interfaces to the audio hardware for capture (microphones) and output (speakers and headphones). Core Audio lets you write applications that work directly with the uncompressed audio data captured from a microphone, perform effects on it, mix it with other audio, and either play the result out to the speakers or convert it into a compressed format that you can then write to the file system or send over the network. If you're not developing full applications, Core Audio lets you write just the custom effect and wrap it in a plug-in called an *audio unit*, which lets users add your effect to their Core Audio-based applications.

Apple debuted Core Audio in Mac OS X 10.0, where it eventually displaced the SoundManager that was part of the Classic Mac OS. Because Core Audio is a C-based API, it can be used with Cocoa applications written in Objective-C and Carbon applications written in C++. You can even skip these application frameworks and call into Core Audio from a plain-C POSIX command-line executable (in fact, most of this book's examples are written this way). Since it is written in and called with C, Core Audio is extremely high-performance, which is crucially important when you're dealing with processing hundreds of thousands of audio samples every second.

Core Audio is based on the idea of "streams" of audio, meaning a continuous series of data that represents an audio signal. Because the sound changes over time, so does the data. Throughout Core Audio, your primary means of interacting with the audio is by working with these streams: getting them from files or input devices, mixing them, converting them to different formats, sending them to output devices, and so on. In doing this, your code makes calls to Core Audio or gets callbacks from Core Audio every time a stream has more data to process. This is a different metaphor than you might have seen in other media APIs. Simple media players such as the HTML5 `<audio>` tag or the iOS `AVAudioPlayer` treat an audio source (such as a file or URL) as an opaque box of

audio: You can usually play, pause, and stop it, and maybe skip ahead or back to different parts of the audio, but you can't really inspect the contents of the box or do anything with the data. What makes Core Audio cool is that it's all about *doing stuff* with the data.

If only it were that easy.

Core Audio has a well-earned reputation as one of the hardest frameworks to deal with on Mac OS X and iPhone. This is because choosing to operate at this level means dealing with a lot of challenges: working with streams of samples in their native form, working with Core Audio's highly asynchronous programming models, and keeping things running fast enough that you don't starve Core Audio when it needs data to send to the speakers or headphones. It didn't help that, in iPhone OS 2.0, the first to support third-party applications, Core Audio was the *only* media framework; developers who simply wanted to play a file had to go all the way down to the stream level to process samples by hand, in C. It's great if you want or need to work with that level, but developers who needed a simpler, higher-level abstraction did a *lot* of public complaining.

Core Audio is not arbitrarily cruel or obtuse. It's complex because the nature of the problem domain is. In our opinion, storing a web app purchase in a database is trivial compared to modeling sound waves in a stream of samples, performing effects on them through mathematical manipulations, and delivering the results to the hardware hundreds or thousands of times a second—and doing it fast enough that the user perceives the result as instantaneous. Doing something really hard, really fast is inherently challenging: By the time you get to the end of this book, we think you'll have an appreciation for just how much Core Audio does for you.

And by that point, we think you'll be ready to do some cool things of your own.

Audience for This Book

One book can't be everything to everyone, so it's best to set the terms right at the start: This book is going to kick your butt. But like Nietzche said, "That which does not kill you only makes you stronger." When you've mastered this material, you'll be ready to do some serious butt-kicking of your own.

Who Should Read this Book

The primary audience for this book is experienced programmers who are familiar with Mac or iOS but have not yet explored Core Audio. Familiarity with C is assumed, but no prerequisite knowledge of digital audio is required; we cover that in Chapter 2. We assume that, to be interested in an audio programming book at all, you've used enough media applications to have a sense of what's possible: audio capture, real-time effects, MP3 playback, virtual instruments, web radio, voice over IP, and so on. If the thought of this stuff doesn't get your programmer parts all tingly, there's probably a nice book on Ruby on Rails two racks over.

Who Shouldn't Read This Book

As self-declared "world's toughest programmer" Mike Lee once said, "Core Audio is some serious black arts shit." You'll find yourself digging around low-level APIs, and if you're not comfortable with getting your hands dirty, this book might not be where you should start (but keep it in mind as something to come back to when you're skilled enough).

You need to know Xcode, C, and Objective-C, and you need to be comfortable reading and interpreting header files. You never know when you'll need to dive deeper into something, and having those skills under your belt will definitely make reading this book a better experience for you.

What You Need to Know

This book assumes a working knowledge of C, including pointers, `malloc()`, and the usual hazards of low-level memory management. If you don't have experience with C or any C-like language (C++, Java, and C#), stop right now and read a good book on C before you attempt to tackle this book.

The book also assumes that you're familiar and comfortable with Xcode and programming in Objective-C. You won't find any primers on how to create a project in Xcode or how to debug; you can find plenty of entry-level books for newbies and converts from other platforms and programming environments. If you're messing around with Core Audio and low-level C APIs, we can assume that you've already got that grounding.

Because the book covers use of Core Audio on the Mac and iOS, we also assume that you have an Apple developer account; if not, you should (they're cheap these days!). Go to **developer.apple.com/mac** or **developer.apple.com/ios** to sign up today—$198 gets you access to all the relevant updates for both Mac OS X and iOS, as well as Xcode, Apple's developer documentation, sample code, and even the session videos from WWDC.

Looking Up Documentation

Every Core Audio developer is constantly flipping over to Core Audio's online documentation to look up function names, parameter lists, and semantic information (such as what you're allowed to pass in a parameter or what to expect a function to do). It's all available on Apple's website, but Apple's online documentation has a habit of moving around, which makes it hard for us to provide URLs that won't break six months after we publish.

Instead, we encourage you to get familiar with Xcode's documentation browser, if you aren't already. In Xcode 4.2, you access it with the **Help > Documentation and API Reference** menu item, which takes you to the Organizer window and opens the Documentation tab. When you first visit this documentation, you'll see a Quick Start

screen that lists some introductory resources for using Xcode. The right pane of this view has buttons for browsing, searching, and managing bookmarks in the documentation. Via Browse, you can select the top-level docsets to work with. Figure I.1 shows the home page for the Mac OS X 10.7 Core Library.

Figure I.1 Xcode documentation viewer showing home page of Mac OS X 10.7 Core Library documentation

The column on the left side of the content pane arranges documentation by type, topic, and then level of the Mac OS X or iOS architecture. If you scroll down to the Media Layer level, you'll find Core Audio, Core Audio Kit, and Audio Toolbox, which is where Core Audio exposes most of its functionality to applications. Click on one of these to see a list of reference guides, technical Q&A documents, and sample code. For example, you could click on Audio Toolbox Framework Reference and then use the Bookmarks toolbar button to find your way back here easily.

Actually, we rarely browse the documentation. The second toolbar button in the left pane exposes a search interface, which is what we use most of the time. Type in a term here to get a list of matching API references (methods, functions, types, structs, and so on), as well as occurrences of the term in other documentation, such as sample code or programming guides, all of which is readable within the documentation viewer. We mentioned the term *audio unit* earlier in the Introduction; Figure I.2 shows what happens when we search for "AudioUnit" with the documentation viewer. As you can see, the term shows up in function names, typedefs, #defines and more in the API section, as well as programming guides, Q&A documents, and sample code in the full-text section.

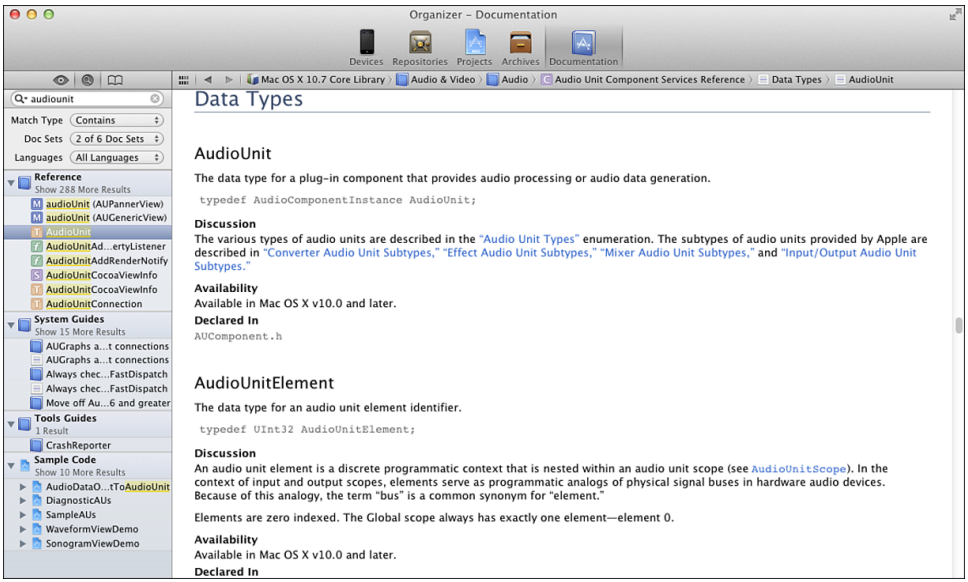

Figure I.2 Searching for "AudioUnit" in Xcode documentation viewer

You can also search for a term directly from your source; just option-double-click on a term in your source to pop up a brief overview of its documentation (see Figure I.3); full documentation is available if you click the book icon at the top of the pop-up window. There's also a button with a **.h** document icon that takes you to the term's definition. Both of these functions are available by Control-clicking (or right-clicking) the term in the text: Look for the menu items Find Text in Documentation *and* Jump to Definition.

Throughout the book, we count on the fact that you can look up information through this interface. For example, when we introduce a new function, we trust you can type its name into the search field and find its API documentation if you want the official word on how it works. When a function uses custom types, these are typically hyperlinked within the documentation so you can follow those links to find related documentation.

How This Book Is Organized

Before you start your journey, let's talk about what's at stake. The path will be treacherous, and you must always remind yourself of the great bounty that awaits you at the end of this book.

We start by giving a high-level overview of what Core Audio does. We briefly describe and provide use cases for the input and output of audio data, "transcoding" between formats, audio effects, playback and recording, and MIDI.

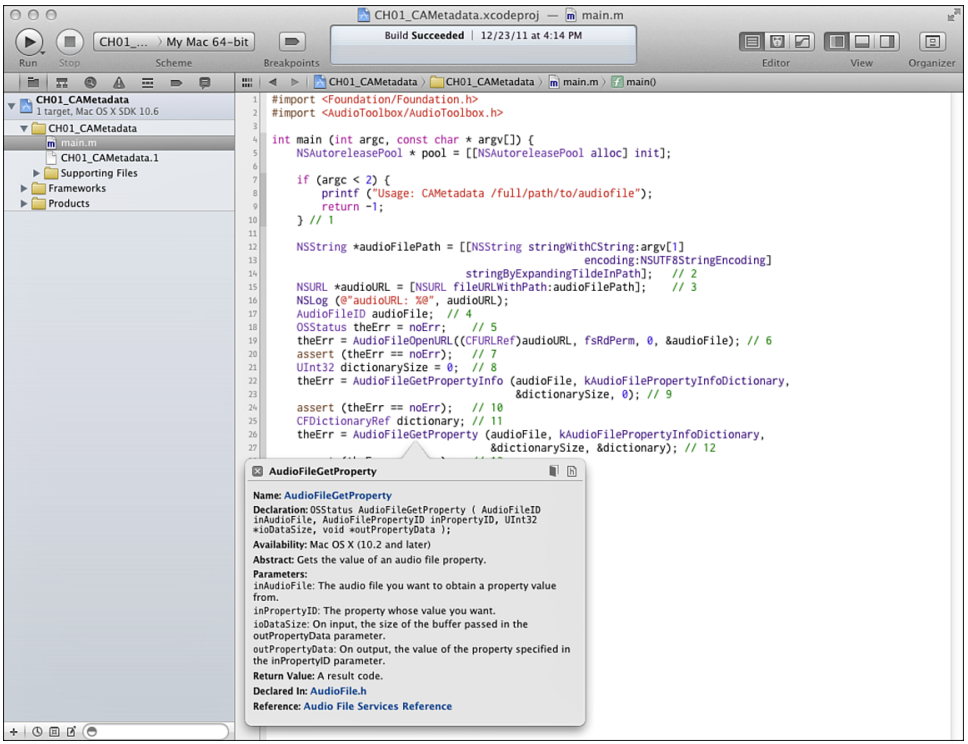

Figure I.3 Looking up documentation from the Xcode source editor

Then we give an overview of the API itself. We describe its procedural, property-driven nature and then give a quick tour of each of its architectural units, starting from high-level API, Audio Queue, OpenAL, Extended Audio File; moving through the mid- and low-level APIs (Audio Units, Audio File, Audio Converter); and finally heading into related topics such as Core MIDI, OpenAL, and how Core Audio works on iOS.

Part I: Understanding Core Audio

This section lays the foundation on which the rest of the book is built. That it seems like a lot of material to get through before writing any code is indicative of the subject. Understanding the problem of digital audio and the solutions Core Audio offers is an absolute must if subsequent sections and their sample code are to make any sense.

- **Chapter 1: Overview of Core Audio**
  We start our journey with a nuts-and-bolts investigation of Core Audio as a Mac or iOS framework: where the files are, how to integrate it into your projects, and where to go for help.

  We start with an overview of the API itself. We describe its procedural, property-driven nature. Then we take a quick tour of each of its architectural units, starting from high-level API (Audio Queue, OpenAL, Extended Audio File), moving

through the midlevel API (Audio Units, Audio File, Audio Converter), and working into the low-level API (IOKit/IOAudio, HAL, and, on iOS, Remote I/O). We also work through a simple application to use Core Audio to fetch metadata from a sound file, to give you a taste of writing Core Audio code.

- **Chapter 2: The Story of Sound**
  The central problem of digital audio lies in representing the analog waveform of a sound on a digital computer. We talk about how sampling converts the sounds you hear into the 1s and 0s of a binary computer. We cover bit rates and the trade-offs of quality, performance, and file size. To get a hands-on understanding of sampling, you'll write your own sound waves directly to a file, sample by sample, and see (well, *hear,* actually) how different wave forms sound different to the human ear.

  When you have the raw stream of bits, you need to quantize them into frames and packets. We talk about the difference between constant and variable bit rates and frame rates.

- **Chapter 3: Audio Processing with Core Audio**
  When you understand the concepts of audio in English, it's time to express them in C. We talk about the implementation details here—how Core Audio represents audio streams and provides functionality to work with those streams. We talk about file formats and stream formats and highlight the difference between them. Then we'll write an example that inspects what kinds of audio format/file format combinations Core Audio supports.

  Following our description of formats, we switch to Core Audio's processing model and look at how Core Audio encapsulates its functionality as Audio Units, how these are combined in interesting ways, and why they use a pull model to move audio data through the system.

Part II: Basic Audio

This section begins the hands-on use of the API and concepts from the previous chapter. We start by discussing the flow of data between files and the audio systems, first by recording and then by playing file-based audio data. Then we discuss transcoding API for moving data between formats and explain the important behind-the-scenes function that serves.

- **Chapter 4: Recording**
  Why address recording before playback? Because it's easier and it generates sample files to play with later. This chapter introduces the high-level API for getting data in and out of files and explores the Audio Queue API for making use of that data. We'll develop a complete example that captures audio from the default input device and writes it to a file. In the process, we'll deal with some of the tricky parts of compressed formats, such as working with format-specific magic cookies and figuring out how big buffers need to be to write data for arbitrary formats.

- **Chapter 5: Playback**
  From a programming perspective, recording and playback are two sides of the same coin. Playback moves data from a file to a series of buffers; recording moves data from a series of buffers into a file.

  This chapter provides a full example of reading audio from a file and playing it to the default output. Again, we look at techniques for dealing with variable-bit-rate formats. We also take a look at some of the neat things you can do with an audio queue's properties and parameters and dig into the latency implications of working with so many buffers.

- **Chapter 6: Conversion**
  For people used to object-oriented programming, Core Audio's high-level API seems pretty low level. This chapter demonstrates the complexity behind the scenes, diving into the nitty-gritty details of modern audio codecs and the complexity necessary to convert them into canonical data. We work through an example that directly uses Audio Converter Services to convert a compressed file to an uncompressed version and then simplify this through the use of Extended Audio File Services, which combine I/O and data conversion into a single step.

Part III: Advanced Audio

Now that you understand how to move audio data back and forth, it's time to get fancy. We start by adding effects to audio data, move into 3D positional audio, and then talk about performance and low-level architecture.

- **Chapter 7: Audio Units: Generators, Effects, and Rendering**
  Core Audio provides an elegant architecture for digital signal processing plug-ins, called Audio Units. However, audio units are the lowest commonly used level of the Core Audio API and introduce new challenges for the programmer. This chapter introduces audio units to play sounds from files and the Mac OS X speech synthesizer and to perform effects on the sound, all coordinated via the AUGraph API. We also look at how to provide your own programmatically generated audio data as input to audio units.

- **Chapter 8: Audio Units: Input and Mixing**
  To help you further flex our muscles with the Audio Units API, we look at how to use the IO unit to perform audio capture and jump through some rather tricky threading hoops to get the captured data to run through an AUGraph of effects and other sources. To combine it all, we make use of the powerful multichannel mixer unit.

- **Chapter 9: Positional Sound**
  Up to now, the discussion has focused on sound itself, but the human experience of sound adds an entirely new dimension to the problem. This section discusses OpenAL, the 3D positional sound API, which enables you to associate sounds with

locations in a 3D space. We start with loops, but by the end of the chapter, you will be able to play arbitrarily long streams of sound from 3D sources.

Part IV: Additional Topics

By this point, we've covered most of Core Audio, but not all of it. This section explores some of the miscellany that doesn't fit into the rest of book. We start with a chapter dedicated to iOS, then talk about handling MIDI data, and end with a chapter on extending Core Audio.

- **Chapter 10: Core Audio on iOS**
  Conceptually, there's little difference between sound on an iPhone and sound on a Macintosh, but the devil is in the details. This chapter addresses the differences, with a particular concentration on the limitations and exceptions that come with limited hardware resources.

  We also discuss the Audio Session API, which is vital to making sure your application behaves properly in the preemptive, multisource, multidestination iPhone environment.

- **Chapter 11: Core MIDI**
  Musicians love the MIDI standard, which is a lynchpin of connecting musical instruments and processing digital music data. In this chapter, we look at how Core MIDI processes music events between the Mac or iOS device and instruments connected either physically or wirelessly. You'll also see how MIDI data can be delivered into an Audio Unit, enabling you to convert note and timing data into sound.

- **Chapter 12: Coda**
  In the final chapter, we look at what we've covered and what's left to discover. We also point out the newest and shiniest audio bits unveiled in Mac OS X 10.7 (Lion) and iOS 5.

This book doesn't use every function Core Audio defines or conceptually cover everything you can do with it, but it does dig deeply into the most used and most important topics. After you make it through the book, you'll have the hang of how Core Audio does things, and we think you'll be well equipped to explore any remaining functionality you need for your applications.

About the Sample Code

The source code for the projects in this book is available on the Resources tab on the book's catalog page:

www.informit.com/title/9780321636843

The downloads contain a README file and folders with the projects for each chapter.

This book contains a lot of sample code, and Core Audio can be pretty verbose to the modern eye. To keep the code size manageable, most of the examples in this book are written as OS X command-line executables, not full-fledged Cocoa (Mac OS X) GUI applications. You'll be able to run these directly in Xcode or from the command line (in Terminal or xterm). The iOS examples in Chapter 10 and thereafter are genuine iOS apps—the iPhone doesn't have a command line, after all—but we don't bother with a GUI unless absolutely necessary.

Core Audio is mostly the same between Mac OS X and iOS, so iOS developers can use the concepts from these examples in iPhone, iPod Touch, and iPad apps. In most cases, the code works exactly the same; we've pointed out any differences between Mac and iOS, either in the APIs themselves or in how they work on the different platforms. For the parts of Core Audio that are unique to iOS, see Chapter 10.

Our baseline SDK for this book is Xcode 4.2, which includes the SDKs for Mac OS X 10.7 (Lion) and iOS 5. For Core Audio development on a Mac, you don't need anything else: All the libraries, headers, and documentation are included with the Xcode tools.[1] Our sample code is written to support versions 10.6 and 10.7 on the Mac, and iOS 4 and 5. Because of changes and deprecations over the years, some of the Mac examples won't run as is on version 10.5 (Leopard) or earlier, although, in many cases, the difference is only a changed constant here or there (for example, the `kAudioFileReadPermission` constant introduced in version 10.6 replaces the `fsRdPerm` found in earlier versions of Mac OS X).

[1] Core Audio used to be a separate download, which has confused a few developers when they've seen it listed separately on Apple's Development Kits page. That download is needed only if you're developing on Tiger (Mac OS X 10.4.*x*).

This page intentionally left blank

1

Overview of Core Audio

Core Audio is the engine behind any sound played on a Mac or iPhone OS. Its procedural API is exposed in C, which makes it directly available in Objective-C and C++, and usable from any other language that can call C functions, such as Java with the Java Native Interface, or Ruby via RubyInline. From an audio standpoint, Core Audio is high level because it is highly agnostic. It abstracts away both the implementation details of the hardware and the details of individual audio formats.

To an application developer, Core Audio is suspiciously low level. If you're coding in C, you're doing something wrong, or so the saying goes. The problem is, very little sits above Core Audio. Audio turns out to be a difficult problem, and all but the most trivial use cases require more decision making than even the gnarliest Objective-C framework. The good news is, the times you don't need Core Audio are easy enough to spot, and the tasks you can do without Core Audio are pretty simple (see sidebar "When Not to Use Core Audio").

When you use Core Audio, you'll likely find it a far different experience from nearly anything else you've used in your Cocoa programming career. Even if you've called into other C-based Apple frameworks, such as Quartz or Core Foundation, you'll likely be surprised by Core Audio's style and conventions.

This chapter looks at what's in Core Audio and where to find it. Then it broadly surveys some of its most distinctive conventions, which you'll get a taste for by writing a simple application to exercise Core Audio's capability to work with audio metadata in files. This will give you your first taste of properties, which enable you to do a lot of the work throughout the book.

When Not to Use Core Audio

The primary scenario for not using Core Audio is when simply playing back from a file: On a Mac, you can use AppKit's `NSSound`, and on iOS, you can use the `AVAudioPlayer` from the AV Foundation framework. iOS also provides the `AVAudioRecorder` for recording to a file. The Mac has no equivalent Objective-C API for recording, although it does have QuickTime and QTKit; you could treat your audio as `QTMovie` objects and pick up some playback, recording, and mixing functionality. However, QuickTime's video orientation and its

> philosophy of being an editing API for multimedia documents makes it a poor fit for purely audio tasks. The same can be said of AV Foundation's `AVPlayer` and `AVCaptureSession` classes, which debuted in iOS 4 and became the heir apparent to QuickTime on Mac in 10.7 (Lion).
>
> Beyond the simplest playback and recording cases—and, in particular, if you want to do anything with the audio, such as mixing, changing formats, applying effects, or working directly with the audio data—you'll want to adopt Core Audio.

The Core Audio Frameworks

Core Audio is a collection of frameworks for working with digital audio. Broadly speaking, you can split these frameworks into two groups: audio engines, which process streams of audio, and helper APIs, which facilitate getting audio data into or out of these engines or working with them in other ways.

Both the Mac and the iPhone have three audio engine APIs:

- **Audio Units.** Core Audio does most of its work in this low-level API. Each unit receives a buffer of audio data from somewhere (the input hardware, another audio unit, a callback to your code, and so on), performs some work on it (such as applying an effect), and passes it on to another unit. A unit can potentially have many inputs and outputs, which makes it possible to mix multiple audio streams into one output. Chapter 7, "Audio Units: Generators, Effects, and Rendering," talks more about Audio Units.
- **Audio Queues.** This is an abstraction atop audio units that make it easier to play or record audio without having to worry about some of the threading challenges that arise when working directly with the time-constrained I/O audio unit. With an audio queue, you record by setting up a callback function to repeatedly receive buffers of new data from the input device every time new data is available; you play back by filling buffers with audio data and handing them to the audio queue. You will do both of these in Chapter 4, "Recording."
- **OpenAL.** This API is an industry standard for creating positional, 3D audio (in other words, surround sound) and is designed to resemble the OpenGL graphics standard. As a result, it's ideally suited for game development. On the Mac and the iPhone, its actual implementation sits atop audio units, but working exclusively with the OpenAL API gets you surprisingly far. Chapter 9, "Positional Sound," covers this in more detail.

To get data into and out of these engines, Core Audio provides various helper APIs, which are used throughout the book:

- **Audio File Services.** This framework abstracts away the details of various container formats for audio files. As a result, you don't have to write code that specifically addresses the idiosyncrasies of AIFFs, WAVs, MP3s, or any other format. It enables your program to open an audio file, get or set the format of the audio data it contains, and start reading or writing.

- **Audio File Stream Services.** If your audio is coming from the network, this framework can help you figure out the format of the audio in the network stream. This enables you to provide it to one of the playback engines or process it in other interesting ways.
- **Audio Converter Services.** Audio can exist in many formats. By the time it reaches the audio engines, it needs to be in an uncompressed playable format (LPCM, discussed in Chapter 2, "The Story of Sound"). Audio Converter Services helps you convert between encoded formats such as AAC or MP3 and the uncompressed raw samples that actually go through the audio units.
- **Extended Audio File Services.** A combination of Audio Converter Services and Audio File Stream Services, the Extended Audio File APIs enables you to read from or write to audio files and do a conversion at the same time. For example, instead of reading AAC data from a file and then converting to uncompressed PCM in memory, you can do both in one call by using Extended Audio File Services.
- **Core MIDI.** Most of the Core Audio frameworks are involved with processing sampled audio that you've received from other sources or captured from an input device. With the Mac-only Core MIDI framework, you synthesize audio on the fly by describing musical notes and how they are to be played out—for example, whether they should sound like they're coming from a grand piano or a ukulele. You'll try out MIDI in Chapter 11, "Core MIDI."

A few Core Audio frameworks are platform specific:

- **Audio Session Services.** This iOS-only framework enables your app to coordinate its use of audio resources with the rest of the system. For example, you use this API to declare an audio "category," which determines whether iPod audio can continue to play while your app plays and whether the ring/silent switch should silence your app. You'll use this more in Chapter 10, "Core Audio on iOS."

As you develop your application, you'll combine these APIs in interesting ways. For example, you could use Audio File Stream Services to get the audio data from a net radio stream and then use OpenAL to put that audio in a specific location in a 3D environment.

Core Audio Conventions

The Core Audio frameworks are exposed as C function calls. This makes them broadly available to Cocoa, Cocoa Touch, and Carbon apps, but you have to be ready to deal with all the usual issues of procedural C, such as pointers, manual memory management, *structs*, and *enums*. Most modern developers have cut their teeth on object-oriented languages such as Objective-C, C++, and Python, so it's no longer a given that professional programmers are comfortable with procedural C.

In C, you don't have classes, object orientation, implementation hiding, or many of the other important language traits that most developers have depended on for years. But Core Audio, like Apple's other C-based frameworks, does provide a measure of these modern traits, even within the C idiom.

Apple's model C framework is Core Foundation, which underlies Foundation, the essential Objective-C framework that nearly all Mac and iPhone applications use. You'll recognize Foundation by classes such as `NSString`, `NSURL`, `NSDate`, and `NSObject`. In many cases, the Objective-C classes assemble their functionality by calling Core Foundation, which provides opaque types (pointers to data structures whose actual members are hidden) and functions that work on these objects. For example, an `NSString` is literally the same as a `CFStringRef` (you can freely cast between the two), and its `length` method is equivalent to the function `CFStringGetLength()`, which takes a `CFStringRef` as its object. By combining these opaque types with consistent naming conventions for functions, Core Foundation provides a highly manageable C API with a clarity similar to what you're used to in Cocoa.

Core Audio is highly similar in its design. Many of its most important objects (such as audio queues and audio files) are treated as opaque objects that you hand to predictably named functions, such as `AudioQueueStart()` or `AudioFileOpenURL()`. It's not explicitly built atop Core Foundation—an `AudioQueueRef` is not technically a CF opaque type; however, it does make use of CF's most important objects, such as `CFStringRef` and `CFURLRef`, which can be trivially cast to and from `NSString`s and `NSURL`s in your Cocoa code.

Your First Core Audio Application

Now let's get a feel for Core Audio code by actually writing some. The audio engine APIs have a lot of moving parts and are, therefore, more complex, so we'll make trivial use of one of the helper APIs. In this first example, we'll get some metadata (information about the audio) from an audio file.

> **Note**
>
> In this book, most of the examples are command-line utilities instead of full-blown AppKit or UIKit applications. This helps keep the focus on the audio code, without bringing in GUI considerations.

Launch Xcode, go to File > New Project, select the Mac OS X Application templates, and choose the Command Line Tool template; select the Foundation type in the pop-up menu below the template icon. When prompted, call the project **CAMetadata**. The resulting project has one user-editable source file, *main.m*, and produces an executable called *CAMetadata*, which you can run from Xcode or in the Terminal.

Select the *CAMetadata.m* file. You'll see that it has a single `main()` function that sets up an `NSAutoReleasePool`, prints a "Hello, World!" log message, and drains the pool

before terminating. Replace the comment and the `printf` so that `main()` looks like Listing 1.1. We've added numbered comments to the ends of some of the statements as callouts so that we can explain what this code does, line by line.

Listing 1.1 **Your First Core Audio Application**

```
int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    if (argc < 2) {
        printf ("Usage: CAMetadata /full/path/to/audiofile\n");
        return -1;
    }                                                                   // 1

NSString *audioFilePath = [[NSString stringWithUTF8String:argv[1]]
                           stringByExpandingTildeInPath];               // 2
    NSURL *audioURL = [NSURL fileURLWithPath:audioFilePath];            // 3
    AudioFileID audioFile;                                              // 4
    OSStatus theErr = noErr;                                            // 5
    theErr = AudioFileOpenURL((CFURLRef)audioURL,
                              kAudioFileReadPermission,
                              0,
                              &audioFile);                              // 6
    assert (theErr == noErr);                                           // 7
    UInt32 dictionarySize = 0;                                          // 8
    theErr = AudioFileGetPropertyInfo (audioFile,
                                       kAudioFilePropertyInfoDictionary,
                                       &dictionarySize,
                                       0);                              // 9
    assert (theErr == noErr);                                           // 10
    CFDictionaryRef dictionary;                                         // 11
    theErr = AudioFileGetProperty (audioFile,
                                   kAudioFilePropertyInfoDictionary,
                                   &dictionarySize,
                                   &dictionary);                        // 12
    assert (theErr == noErr);                                           // 13
    NSLog (@"dictionary: %@", dictionary);                              // 14
    CFRelease (dictionary);                                             // 15
    theErr = AudioFileClose (audioFile);                                // 16
    assert (theErr == noErr);                                           // 17

    [pool drain];
    return 0;
}
```

Now let's walk through the code from Listing 1.1:

1. As in any C program, the `main()` method accepts a count of arguments (`argc`) and an array of plain C-string arguments. The first string is the executable name, so you must look to see if there's a second argument that provides the path to an audio file. If there isn't, you print a message and terminate.
2. If there's a path, you need to convert it from a plain C string to the `NSString`/`CFStringRef` representation that Apple's various frameworks use. Specifying the UTF-8 encoding for this conversion lets you pass in paths that use non-Western characters, in case (like us) you listen to music from all over the world. By using `stringByExpandingTildeInPath`, you accept the tilde character as a shortcut to the user's home directory, as in *~/Music/*….
3. The Audio File APIs work with URL representations of file paths, so you must convert the file path to an `NSURL`.
4. Core Audio uses the `AudioFileID` type to refer to audio file objects, so you declare a reference as a local variable.
5. Most Core Audio functions signal success or failure through their return value, which is of type `OSStatus`. Any status other than `noErr` (which is `0`) signals an error. You need to check this return value on *every* Core Audio call because an error early on usually makes subsequent calls meaningless. For example, if you can't create the `AudioFileID` object, trying to get properties from the file that object was supposed to represent will always fail. In this example, we've used an `assert()` to terminate the program instantly if we ever get an error, in callouts 7, 10, 13, and 17. Of course, your application will probably want to handle errors with somewhat less brutality.
6. Here's the first Core Audio function call: `AudioFileOpenURL`. It takes four parameters, a `CFURLRef`, a file permissions flag, a file type hint, and a pointer to receive the created `AudioFileID` object. You do a toll-free cast of the `NSURL` to a `CFURLRef` to match the first parameter's defined type. For the file permissions, you pass a constant to indicate read permission. You don't have a hint to provide, so you pass 0 to make Core Audio figure it out for itself. Finally, you use the `&` ("address of") operator to provide a pointer to receive the `AudioFileID` object that gets created.
7. If `AudioFileOpenURL` returned an error, die.

8. To get the file's metadata, you will be asking for a metadata property, `kAudioFilePropertyInfoDictionary`. But that call requires allocating memory for the returned metadata in advance. So here, we declare a local variable to receive the size we'll need to allocate.
9. To get the needed size, call `AudioFileGetPropertyInfo`, passing in the `AudioFileID`, the property you want information about, a pointer to receive the result, and a pointer to a flag variable that indicates whether the property is writeable (because we don't care, we pass in `0`).
10. If `AudioFileGetPropertyInfo` failed, terminate.
11. The call to get a property from an audio file populates different types, based on the property itself. Some properties are numeric; some are strings. The documentation and the Core Audio header files describe these values. Asking for `kAudioFilePropertyInfoDictionary` results in a dictionary, so we set up a local variable instance of type `CFDictionaryRef` (which can be cast to an `NSDictionary` if needed).
12. You're finally ready to request the property. Call `AudioFileGetProperty`, passing in the `AudioFileID`, the property constant, a pointer to the size you're prepared to accept (set up in callouts 8–10 with the `AudioFileGetPropertyInfo` call) and a pointer to receive the value (set up on the previous line).
13. Again, check the return value and fail if it's anything other than `noErr`.
14. Let's see what you got. As in any Core Foundation or Cocoa object, you can use `"%@"` in a format string to get a string representation of the dictionary.
15. Core Foundation doesn't offer autorelease, so the `CFDictionaryRef` received in callout 12 has a retain count of 1. `CFRelease()` releases your interest in the object.
16. The `AudioFileID` also needs to be cleaned up but isn't a Core Foundation object, per se; therefore, it doesn't get `CFRelease()`'d. Instead, it has its own end-of-life function: `AudioFileClose()`.
17. `AudioFileClose()` is another Core Audio call, so you should continue to check return codes, though it's arguably meaningless here because you're two lines away from terminating anyway.

So that's about 30 lines of code, but functionally, it's all about setting up three calls: opening a file, allocating a buffer for the metadata, and getting the metadata.

Running the Example

That was probably more code than you're used to writing for simple functionality, but it's done now. Let's try it out. Click build; you get compile errors. Upon inspection, you should see that all the Core Audio functions and constants aren't being found.

This is because Core Audio isn't included by default in Xcode's command-line executable project, which imports only the Foundation framework. Add a second `#import` line:

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

Audio Toolbox is an "umbrella" header file that includes most of the Core Audio functionality you'll use in your apps, which means you'll be importing it into pretty much all the examples. You also need to add the framework to your project. Click the project icon in Xcode's file navigator, select the CAMetadata target, and click the Build Phases tab. Expand the Link Binaries with Libraries section and click the + button to add a new library to be linked at build time. In the sheet that slides out, select the `AudioToolbox.framework`, as shown in Figure 1.1.

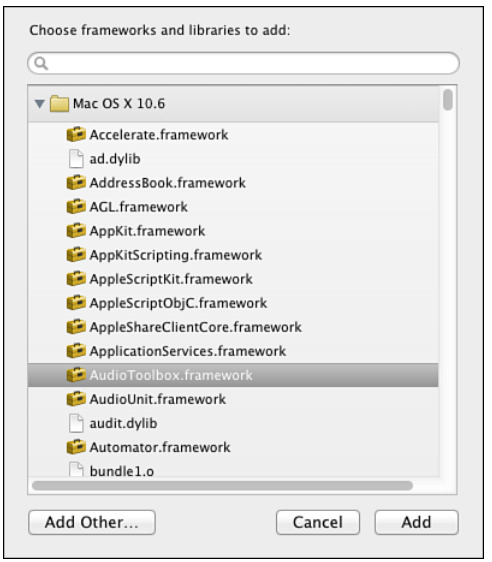

Figure 1.1 Adding the AudioToolbox.framework to an Xcode project

Now you should be able to build the application without any errors. To run it, you need to provide a path as a command-line argument. You can either open the Terminal and navigate to the project's build directory or supply an argument with Xcode. Let's do the latter:

- From the Scheme pop-up menu, select Edit Scheme.
- Select the Run CAMetadata item and click the Arguments tab.
- Press + to add an argument and supply the path to an audio file on your hard drive.

- If your path has spaces in it, use quotation marks. For example, we're using an MP3 bought online, located at *~/Music/iTunes/iTunes Music/Amazon MP3/Metric/Fantasies/05 - Gold Guns Girls.mp3*. Click OK to dismiss the Scheme Editor sheet.
- Bring up Xcode's Console pane with Shift-„-C and click Run.

Assuming that your path is valid, your output will look something like this:

```
2010-02-18 09:43:17.623 CAMetadata[17104:a0f] dictionary: {
    album = Fantasies;
    "approximate duration in seconds" = "245.368";
    artist = Metric;
    comments = "Amazon.com Song ID: 210266948";
    copyright = "2009 Metric Productions";
    genre = "Alternative Rock";
    title = "Gold Guns Girls";
    "track number" = "5/10";
    year = 2009;
}
```

Well, that's pretty cool: You've got a nice dump of a lot of the same metadata that you'd see in an application such as iTunes. Now let's check it out with an AAC song from the iTunes Store. Changing the command-line argument to something like *~/Music/iTunes/iTunes Music/Arcade Fire/Funeral/07 Wake Up.m4a* gets you the following on Snow Leopard:

```
2010-02-18 09:48:15.421 CAMetadata[17665:a0f] dictionary: {
    "approximate duration in seconds" = "335.333";
}
```

Whoa! What happened to the metadata call?

Nothing, really: Nothing in the documentation promises what you can expect in the info dictionary. As it turns out, Core Audio offers richer support for ID3 tags in *.mp3* files than the iTunes tagging found in *.m4a* files.

No Need for Promises

Speaking from experience, you'll want to prepare yourself for unpredictable results, such as different levels of metadata support for MP3 and AAC files. Mastering Core Audio isn't just about understanding the APIs; it's also about developing a sense of the implementation, how the library actually works, and what it does well and where it comes up short.

Core Audio isn't just about the syntax of your calls; it's about the semantics, too. In some cases, code that's syntactically correct will fail in practice because it violates implicit contracts, acts differently on different hardware, or even just it uses too much CPU time in a time-constrained callback. The successful Core Audio programmer doesn't march off in a huff when things don't work as expected or don't work well enough the first time. Instead, you must try to figure out what's really going on and come up with a better approach.

Core Audio Properties

The Core Audio calls in this example were all about getting properties from the audio file object. The routine of preparing for and executing property-setter and property-getter calls is essential in Core Audio.

That's because Core Audio is a *property-driven* API. Properties are key-value pairs, with the keys being enumerated integers. The values can be of whatever type the API defines. Each API in Core Audio communicates its capabilities and state via its list of properties. For example, if you look up the `AudioFileGetProperty()` function in this example, you'll find a link to a list of Audio File Properties in the documentation. The list, which you can also find by looking in Core Audio's *AudioFile.h* header, looks like this:

```
kAudioFilePropertyFileFormat            =    'ffmt',
kAudioFilePropertyDataFormat            =    'dfmt',
kAudioFilePropertyIsOptimized           =    'optm',
kAudioFilePropertyMagicCookieData       =    'mgic',
kAudioFilePropertyAudioDataByteCount    =    'bcnt',
...
```

These keys are 32-bit integer values that you can read in the documentation and header file as four character codes. As you can see from this list, the four-character codes take advantage of the fact that you can use single quotes to represent char literals in C and spell out clever mnemonics. Assume that `fmt` is short for "format," and you can figure out that `ffmt` is the code for "file format" and `dfmt` means "data format." Codes like these are used throughout Core Audio, as property keys and sometimes as error statuses. If you attempt to write to a file format Core Audio doesn't understand, you'll get the response `fmt?`, which is `kAudioFileUnsupportedDataFormatError`.

Because Core Audio makes so much use of properties, you'll see common patterns throughout its API for setting and getting properties. You've already seen `AudioFileGetPropertyInfo()` and `AudioFileGetProperty()`, so it probably won't surprise you later to encounter `AudioQueueGetProperty()`, `AudioUnitGetProperty()`, `AudioConverterGetProperty()`, and so on. Some APIs provide property listeners that you can register to receive a callback when a property changes. Using callback functions to respond to asynchronous events is a common pattern in Core Audio.

The values that you get or set with these APIs depend on the property being set. You retrieved the `kAudioFilePropertyInfoDictionary` property, which returned a pointer to a `CFDictionaryRef`, but if you had asked for a `kAudioFilePropertyEstimatedDuration`, you'd need to be prepared to accept a pointer to an `NSTimeInterval` (which is really just a `double`). This is tremendously powerful because a small number of functions can support a potentially infinite set of properties. However, setting up such calls does involve extra work because you typically have to use the "get property info" call to allocate some memory to receive the property value or to inspect whether the property is writable.

Another point to notice with the property functions is the Core Audio naming conventions for function parameters. Let's look at the definition of `AudioFileGetProperty()` from the docs (or the *AudioFile.h* header):

```
OSStatus AudioFileGetProperty (
   AudioFileID         inAudioFile,
   AudioFilePropertyID inPropertyID,
   UInt32              *ioDataSize,
   void                *outPropertyData
);
```

Notice the names of the parameters: The use of `in`, `out`, or `io` indicates whether a parameter is used only for input to the function (as with the first two, which indicate the file to use and the property you want), only for output from the function (as with the fourth, `outPropertyData`, which fills a pointer with the property value), or for input and output (as with the third, `ioDataSize`, which accepts the size of buffer you allocated for `outPropertyData` and then writes back the number of bytes actually written into that buffer). You'll see this naming pattern throughout Core Audio, particularly any time a parameter works with a pointer to populate a value.

Summary

This chapter provided an overview of the many different parts of Core Audio and gave you a taste of programming by using Audio File Services to get the metadata properties of audio files on the local drive. You saw how Core Audio uses properties as a crucial idiom for working with its various APIs. You also saw how Core Audio uses four character codes to specify property keys, and to signal errors.

Of course, you haven't really dealt with audio itself yet. To do that, you first need to understand how sound is represented and handled in a digital form. Then you'll be ready to dig into Core Audio's APIs for working with audio data.

This page intentionally left blank

Index

Numbers

A

B

C

D

E

F

G–H

I

J–K–L

M

N

O

P

Q–R

S

T

U–Z