Erica Sadun

Companion to *The Core iOS 6 Developer's Cookbook*

The Advanced iOS 6 Developer's Cookbook

Developer's Library

FREE SAMPLE CHAPTER

The Advanced iOS 6 Developer's Cookbook

Fourth Edition

Erica Sadun

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

ISBN-13: 978-0-321-88422-0
ISBN-10: 0-321-88422-1

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Jovana San Nicolas-Shirley

Copy Editor
Apostrophe Editing Services

Indexer
Brad Herriman

Proofreader
Sarah Kearns

Technical Editors
Mike Shields
Rich Wardwell

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

❖

I dedicate this book with love to my husband, Alberto,
who has put up with too many gadgets and
too many SDKs over the years while remaining
both kind and patient at the end of the day.

❖

Contents at a Glance

Table of Contents

Acknowledgments

This book would not exist without the efforts of Chuck Toporek, who was my editor and whipcracker for many years and multiple publishers. He is now at Apple and deeply missed. There'd be no Cookbook were it not for him. He balances two great skill sets: inspiring authors to do what they think they cannot, and wielding the large "reality trout" of whacking[1] to keep subject matter focused and in the real world. There's nothing like being smacked repeatedly by a large virtual fish to bring a book in on deadline and with compelling content.

Thanks go as well to Trina MacDonald (my terrific new editor), Chris Zahn (the awesomely talented development editor), and Olivia Basegio (the faithful and rocking editorial assistant who kept things rolling behind the scenes). Also, a big thank you to the entire Addison-Wesley/Pearson production team, specifically Kristy Hart, Jovana San Nicolas-Shirley, San Dee Phillips, Nonie Ratcliff, and Chuti Prasertsith. Thanks also to the crew at Safari for getting my book up in Rough Cuts and for quickly fixing things when technical glitches occurred.

Thanks go as well to Neil Salkind, my agent of many years, to the tech reviewers Oliver Drobnik, Rich Wardwell, and Duncan Champney, who helped keep this book in the realm of sanity rather than wishful thinking, and to all my colleagues, both present and former, at TUAW, Ars Technica, and the Digital Media/Inside iPhone blog.

I am deeply indebted to the wide community of iOS developers, including Jon Bauer, Tim Burks, Matt Martel, Tim Isted, Joachim Bean, Aaron Basil, Roberto Gamboni, John Muchow, Scott Mikolaitis, Alex Schaefer, Nick Penree, James Cuff, Jay Freeman, Mark Montecalvo, August Joki, Max Weisel, Optimo, Kevin Brosius, Planetbeing, Pytey, Michael Brennan, Daniel Gard, Michael Jones, Roxfan, MuscleNerd, np101137, UnterPerro, Jonathan Watmough, Youssef Francis, Bryan Henry, William DeMuro, Jeremy Sinclair, Arshad Tayyeb, Jonathan Thompson, Dustin Voss, Daniel Peebles, ChronicProductions, Greg Hartstein, Emanuele Vulcano, Sean Heber, Josh Bleecher Snyder, Eric Chamberlain, Steven Troughton-Smith, Dustin Howett, Dick Applebaum, Kevin Ballard, Hamish Allan, Lutz Bendlin, Oliver Drobnik, Rod Strougo, Kevin McAllister, Jay Abbott, Tim Grant Davies, Maurice Sharp, Chris Samuels, Chris Greening, Jonathan Willing, Landon Fuller, Jeremy Tregunna, Christine Reindl, Wil Macaulay, Stefan Hafeneger, Scott Yelich, Mike Kale, chrallelinder, John Varghese, Robert Jen, Andrea Fanfani, J. Roman, jtbandes, Artissimo, Aaron Alexander, Christopher Campbell Jensen, Nico Ameghino, Jon Moody, Julián Romero, Scott Lawrence, Evan K. Stone, Kenny Chan Ching-King, Matthias Ringwald, Jeff Tentschert, Marco Fanciulli, Neil Taylor, Sjoerd van Geffen, Absentia, Nownot, Emerson Malca, Matt Brown, Chris Foresman, Aron Trimble, Paul Griffin, Paul Robichaux, Nicolas Haunold, Anatol Ulrich (hypnocode GmbH), Kristian Glass, Remy "psy" Demarest, Yanik Magnan, ashikase, Shane Zatezalo, Tito Ciuro, Mahipal Raythattha, Jonah Williams of Carbon Five, Joshua Weinberg, biappi, Eric Mock, and everyone at the iPhone developer channels at irc.saurik.com and irc.freenode.net, among many others too numerous to name individually. Their techniques, suggestions, and feedback helped make this book possible. If I have overlooked anyone who helped contribute, please accept my apologies for the oversight.

Special thanks go out to my family and friends, who supported me through month after month of new beta releases and who patiently put up with my unexplained absences and frequent howls of despair. I appreciate you all hanging in there with me. And thanks to my children for their steadfastness, even as they learned that a hunched back and the sound of clicking keys is a pale substitute for a proper mother. My kids provided invaluable assistance over the past few months by testing applications, offering suggestions, and just being awesome people. I try to remind myself on a daily basis how lucky I am that these kids are part of my life.

About the Author

Erica Sadun is the bestselling author, coauthor, and contributor to several dozen books on programming, digital video and photography, and web design, including the widely popular *The iOS 5 Developer's Cookbook*. She currently blogs at TUAW.com and has blogged in the past at O'Reilly's Mac Devcenter, Lifehacker, and Ars Technica. In addition to being the author of dozens of iOS-native applications, Erica holds a Ph.D. in Computer Science from Georgia Tech's Graphics, Visualization, and Usability Center. A geek, a programmer, and an author, she's never met a gadget she didn't love. When not writing, she and her geek husband parent three geeks-in-training, who regard their parents with restrained bemusement, when they're not busy rewiring the house or plotting global dominance.

Preface

Welcome to another iOS Cookbook!

With iOS 6, Apple's mobile device family has reached new levels of excitement and possibility. This Cookbook is here to help you start developing. This revision introduces new features announced at the latest WWDC, showing you how to incorporate them into your applications.

For this edition, my publishing team has sensibly split the Cookbook material into manageable print volumes. This book, *The Advanced iOS 6 Developer's Cookbook,* centers on common frameworks such as StoreKit, GameKit, and Core Location and handy techniques such as image manipulation typesetting. It helps you build applications that leverage special-purpose libraries and move beyond the basics. This volume is for those who have a strong grasp on iOS development and are looking for practical how-to's for specialized areas.

Its companion volume, *The Core iOS 6 Developer's Cookbook,* provides solutions for the heart of day-to-day development. It covers all the classes you need for creating iOS applications using standard APIs and interface elements. It contains the recipes you need for working with graphics, touches, and views to create mobile applications.

Finally, there's *Learning iOS 6: A Hands-On Guide to the Fundamentals of iOS Programming*, which covers much of the tutorial material that used to compose the first several chapters of the Cookbook. There you can find all the fundamental how-to's you need to learn iOS 6 development from the ground up. From Objective-C to Xcode, debugging to deployment, *Learning iOS 6* teaches you how to start with Apple's development tool suite.

As in the past, you can find sample code at github. You'll find the repository for this Cookbook at https://github.com/erica/iOS-6-Cookbook, all of it refreshed for iOS 6 after WWDC 2012.

If you have suggestions, bug fixes, corrections, or anything else you'd like to contribute to a future edition, please contact me at erica@ericasadun.com. Let me thank you all in advance. I appreciate all feedback that helps make this a better, stronger book.

—Erica Sadun, September 2012

What You Need

It goes without saying that, if you plan to build iOS applications, you need at least one iOS device to test your application, preferably a new model iPhone or tablet. The following list covers the basics of what you need to begin:

- **Apple's iOS SDK**—You can download the latest version of the iOS SDK from Apple's iOS Dev Center (http://developer.apple.com/ios). If you plan to sell apps through the App Store, become a paid iOS developer. This costs $99/year for individuals and $299/year for enterprise (that is, corporate) developers. Registered developers receive certificates that enable them to "sign" and download their applications to their iPhone/iPod touch for testing and debugging and to gain early access to prerelease versions of iOS. Free-program

developers can test their software on the Mac-based simulator but cannot deploy to devices or submit to the App Store.

> **University Student Program**
>
> Apple also offers a University Program for students and educators. If you are a computer science student taking classes at the university level, check with your professor to see whether your school is part of the University Program. For more information about the iPhone Developer University Program, see http://developer.apple.com/support/iphone/university.

- **A modern Mac running Mac OS X Lion (v 10.7) or, preferably, Mac OS X Mountain Lion (v 10.8)**—You need plenty of disk space for development, and your Mac should have as much RAM as you can afford to put into it.
- **An iOS device**—Although the iOS SDK includes a simulator for you to test your applications in, you really do need to own iOS hardware to develop for the platform. You can tether your unit to the computer and install the software you've built. For real-life App Store deployment, it helps to have several units on hand, representing the various hardware and firmware generations, so you can test on the same platforms your target audience uses.
- **An Internet connection**—This connection enables you to test your programs with a live Wi-Fi connection and with an EDGE or 3G service.
- **Familiarity with Objective-C**—To program for the iPhone, you need to know Objective-C 2.0. The language is based on ANSI C with object-oriented extensions, which means you also need to know a bit of C. If you have programmed with Java or C++ and are familiar with C, you can make the move to Objective-C.

Your Roadmap to Mac/iOS Development

One book can't be everything to everyone. Try as I might, if we were to pack everything you need to know into this book, you wouldn't be able to pick it up. (As it stands, this book offers an excellent tool for upper-body development. Please don't sue if you strain yourself lifting it.) There is, indeed, a lot you need to know to develop for the Mac and iOS platforms. If you are just starting out and don't have any programming experience, your first course of action should be to take a college-level course in the C programming language. Although the alphabet might start with the letter A, the root of most programming languages, and certainly your path as a developer, is C.

When you know C and how to work with a compiler (something you'll learn in that basic C course), the rest should be easy. From there, you can hop right on to Objective-C and learn how to program with that alongside the Cocoa frameworks. The flowchart shown in Figure P-1 shows you key titles offered by Pearson Education that can help provide the training you need to become a skilled iOS developer.

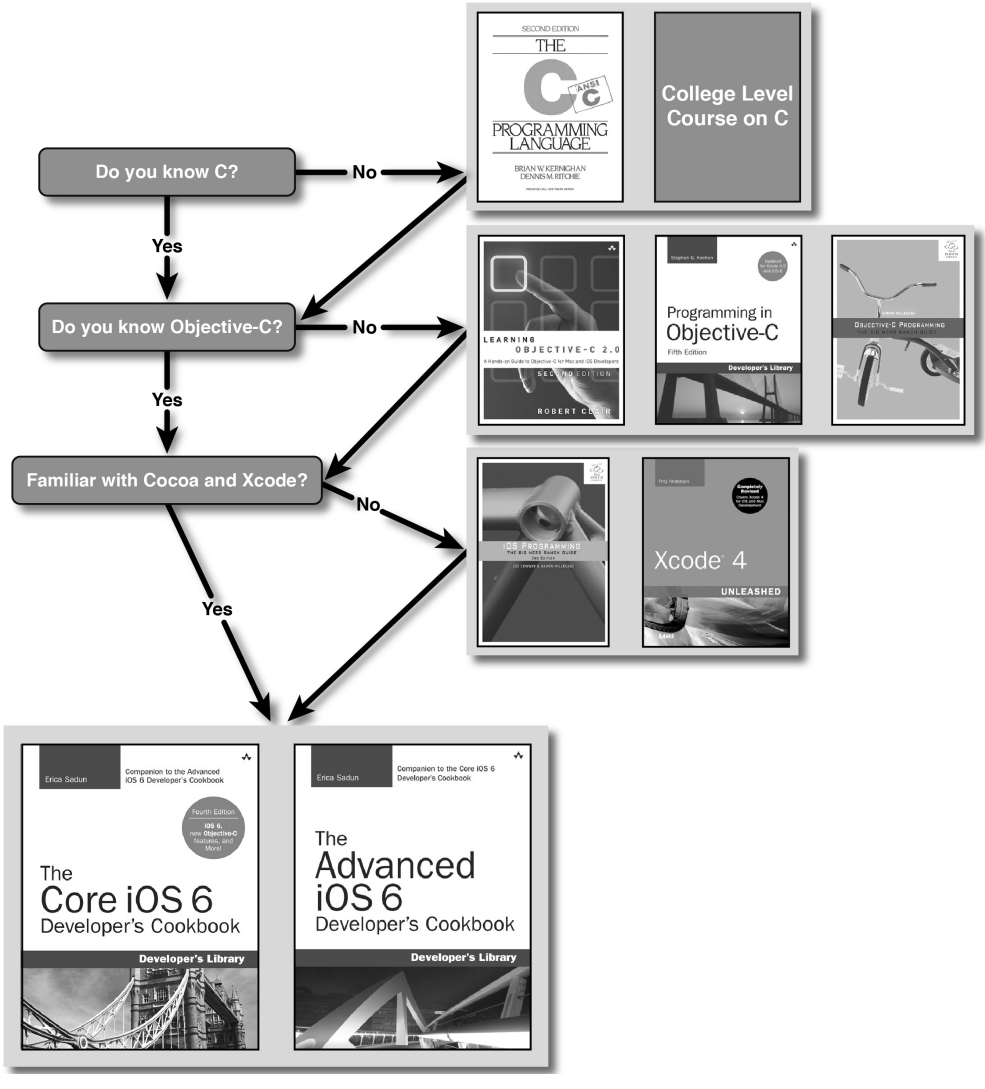

Figure P-1 A roadmap to becoming an iOS developer

When you know C, you have a few options for learning how to program with Objective-C. If you want an in-depth view of the language, you can either read Apple's documentation or pick up one of these books on Objective-C:

- *Objective-C Programming: The Big Nerd Ranch Guide,* by Aaron Hillegass (Big Nerd Ranch, 2012)

- *Learning Objective-C: A Hands-on Guide to Objective-C for Mac and iOS Developers,* by Robert Clair (Addison-Wesley, 2011)
- *Programming in Objective-C 2.0, Fourth Edition,* by Stephen Kochan (Addison-Wesley, 2012)

With the language behind you, next up is tackling Cocoa and the developer tools, otherwise known as Xcode. For that, you have a few different options. Again, you can refer to Apple's documentation on Cocoa and Xcode,[2] or if you prefer books, you can learn from the best. Aaron Hillegass, founder of the Big Nerd Ranch in Atlanta,[3] is the coauthor of *iOS Programming: The Big Nerd Ranch Guide, Second Edition,* and author of *Cocoa Programming for Mac OS X,* soon to be in its fourth edition. Aaron's book is highly regarded in Mac developer circles and is the most-recommended book you'll see on the cocoa-dev mailing list. To learn more about Xcode, look no further than Fritz Anderson's *Xcode 4 Unleashed* from Sams Publishing.

> **Note**
>
> There are plenty of other books from other publishers on the market, including the bestselling *Beginning iPhone 4 Development*, by Dave Mark, Jack Nutting, and Jeff LaMarche (Apress, 2011). Another book that's worth picking up if you're a total newbie to programming is *Beginning Mac Programming*, by Tim Isted (Pragmatic Programmers, 2011). Don't just limit yourself to one book or publisher. Just as you can learn a lot by talking with different developers, you can learn lots of tricks and tips from other books on the market.

To truly master Mac development, you need to look at a variety of sources: books, blogs, mailing lists, Apple's documentation, and, best of all, conferences. If you get the chance to attend WWDC, you'll know what I'm talking about. The time you spend at those conferences talking with other developers, and in the case of WWDC, talking with Apple's engineers, is well worth the expense if you are a serious developer.

How This Book Is Organized

This book offers single-task recipes for the most common issues new iOS developers face: laying out interface elements, responding to users, accessing local data sources, and connecting to the Internet. Each chapter groups together related tasks, enabling you to jump directly to the solution you're looking for without having to decide which class or framework best matches that problem.

The iOS 6 Developer's Cookbook offers you "cut-and-paste convenience," which means you can freely reuse the source code from recipes in this book for your own applications and then tweak the code to suit your app's needs.

Here's a rundown of this book's chapters:

- **Chapter 1, "Device-Specific Development"**—Each iOS device represents a meld of unique, shared, momentary, and persistent properties. These properties include the device's current physical orientation, its model name, its battery state, and its access to

onboard hardware. This chapter looks at the device from its build configuration to its active onboard sensors. It provides recipes that return a variety of information items about the unit in use.

- **Chapter 2, "Documents and Data Sharing"**—Under iOS, applications can share information and data as well as move control from one application to another using several system-supplied features. This chapter introduces the ways you can integrate documents and data sharing between applications. You see how to add these features into your applications and use them smartly to make your app a cooperative citizen of the iOS ecosystem.
- **Chapter 3, "Core Text"**—This chapter introduces attributed text processing and explores how you can build text features into your apps. You read about adding attributed strings to common UIKit elements, how to create Core Text-powered views, and how to break beyond lines for freeform text typesetting. After reading this chapter, you'll have discovered the power that Core Text brings to iOS.
- **Chapter 4, "Geometry"**—Although UIKit requires less applied math than, say, Core Animation or Open GL, geometry plays an important role when working with Bezier paths and view transforms. Why do you need geometry? It helps you manipulate views in nonstandard ways, including laying out text along custom paths and performing path-follow types of animation. If your eyes glaze over at the mention of Bezier curves, Convex Hulls, and splines, this chapter can help demystify these terms, enabling you to add some powerful customization options to your toolbox.
- **Chapter 5, "Networking"**—Apple has lavished iOS with a solid grounding in all kinds of network computing and its supporting technologies. The networking chapter in the Core Cookbook introduced network status checks, synchronous and asynchronous downloads, JSON, and XML parsing. This chapter continues that theme by introducing more advanced techniques. These include authentication challenges, using the system keychain, working with OAuth, and so forth. Here are handy approaches that should help with your development.
- **Chapter 6, "Images"**—Images are abstract representations, storing data that makes up pictures. This chapter introduces Cocoa Touch images, specifically the UIImage class, and teaches you all the basic know-how you need for working with image data on iOS. In this chapter, you learn how to load, store, and modify image data in your applications. You discover how to process image data to create special effects, how to access images on a byte-by-byte basis, and more.
- **Chapter 7, "Camera"**—Cameras kick images up to the next level. They enable you to integrate live feeds and user-directed snapshots into your applications, and provide raw data sourced from the real world. In this chapter, you read about image capture. You discover how to take pictures using Apple-sourced classes and how to roll your own from scratch. You learn about controlling image metadata and how to integrate live feeds with advanced filtering. This chapter focuses on image capture from a hardware point of view. Whether you're switching on the camera flash LED or detecting faces, this chapter introduces the ins and outs of iOS image capture technology.

- **Chapter 8, "Audio"**—The iOS device is a media master; its built-in iPod features expertly handle both audio and video. The iOS SDK exposes that functionality to developers. A rich suite of classes simplifies media handling via playback, search, and recording. This chapter introduces recipes that use those classes for audio, presenting media to your users and letting your users interact with that media. You see how to build audio players and audio recorders. You discover how to browse the iPod library and how to choose what items to play.
- **Chapter 9, "Connecting to the Address Book"**—This chapter introduces the Address Book and demonstrates how to use its frameworks in your applications. You read about accessing information on a contact-by-contact basis, how to modify and update contact information, and how to use predicates to find just the contact you're interested in. This chapter also covers the GUI classes that provide interactive solutions for picking, viewing, and modifying contacts.
- **Chapter 10, "Location"**—Where you compute is fast becoming just as important as how you compute and what you compute. iOS is constantly on the go, traveling with its users throughout the course of the day. Core Location infuses iOS with on-demand geopositioning. MapKit adds interactive in-application mapping, enabling users to view and manipulate annotated maps. With Core Location and MapKit, you can develop applications that help users meet up with friends, search for local resources, or provide location-based streams of personal information. This chapter introduces these location-aware frameworks and shows you how you can integrate them into your iOS applications.
- **Chapter 11, "GameKit"**—This chapter introduces various ways you can create connected game play through GameKit. GameKit offers features that enable your applications to move beyond a single-player/single-device scenario toward using Game Center and device-to-device networking. Apple's Game Center adds a centralized service that enables your game to offer shared leaderboards and Internet-based matches. GameKit also provides an ad-hoc networking solution for peer-to-peer connectivity.
- **Chapter 12, "StoreKit"**—StoreKit offers in-app purchasing that integrates into your software. With StoreKit, end users can use their iTunes credentials to buy unlockable features, media subscriptions, or consumable assets, such as fish food or sunlight, from within an application. This chapter introduces StoreKit and shows you how to use the StoreKit API to create purchasing options for users.
- **Chapter 13, "Push Notification"**—When off-device services need to communicate directly with users, push notifications provide a solution. Just as local notifications enable apps to contact users at scheduled times, push notifications deliver messages from web-based systems. Push notifications enable devices to display an alert, play a custom sound, or update an application badge. In this way, off-device services connect with an iOS-based client, enabling them to know about new data or updates. This chapter introduces all the push notification basics you need to know.

About the Sample Code

For the sake of pedagogy, this book's sample code uses a single main.m file. This is not how people normally develop iPhone or Cocoa applications, or, honestly, how they should be developing them, but it provides a great way of presenting a single big idea. It's hard to tell a story when readers must look through five or seven or nine individual files at once. Offering a single file concentrates that story, allowing access to that idea in a single chunk.

These examples are not intended as stand-alone applications. They are there to demonstrate a single recipe and a single idea. One main.m file with a central presentation reveals the implementation story in one place. You can study these concentrated ideas and transfer them into normal application structures, using the standard file structure and layout. The presentation in this book does not produce code in a standard day-to-day best-practices approach. Instead, it offers concise solutions that you can incorporate back into your work as needed.

Contrast that to Apple's standard sample code, where you must comb through many files to build up a mental model of the concepts that are being demonstrated. Those examples are built as full applications, often doing tasks that are related to but not essential to what you need to solve. Finding just those relevant portions is a lot of work. The effort may outweigh any gains.

In this book, you find exceptions to this one-file-with-the-story rule: The Cookbook provides standard class and header files when a class implementation is the recipe. Instead of highlighting a technique, some recipes offer these classes and categories (that is, extensions to a preexisting class rather than a new class). For those recipes, look for separate .m *and* .h files in addition to the skeletal main.m that encapsulates the rest of the story.

For the most part, the examples for this book use a single application identifier: com.sadun.helloworld. This book uses one identifier to avoid clogging up your iOS devices with dozens of examples at once. Each example replaces the previous one, ensuring that your home screen remains relatively uncluttered. If you want to install several examples simultaneously, simply edit the identifier, adding a unique suffix, such as com.sadun.helloworld.table-edits. You can also edit the custom display name to make the apps visually distinct. Your Team Provisioning Profile matches every application identifier, including com.sadun.helloworld. This enables you to install compiled code to devices without having to change the identifier; just make sure to update your signing identity in each project's build settings.

Getting the Sample Code

You'll find the source code for this book at github.com/erica/iOS-6-Cookbook on the open-source GitHub hosting site. There, you can find a chapter-by-chapter collection of source code that provides working examples of the material covered in this book. Recipes are numbered as they are in the book. Recipe 6 in Chapter 5, for example, appears in the C05 folder in the 06 subfolder.

Any project numbered 00 or that has a suffix (such as 05b or 02c) refers to material used to create in-text coverage and figures. Normally I delete these extra projects. Early readers of this

manuscript requested that I include them in this edition. You can find a half dozen or so of these extra samples scattered around the repository.

If you do not feel comfortable using git directly, GitHub offers a download button. It was at the right side of the main page at the time this book was written, about halfway down the first page. It enables you to retrieve the entire repository as a ZIP archive or tarball.

Contribute!

Sample code is never a fixed target. It continues to evolve as Apple updates its SDK and the Cocoa Touch libraries. Get involved. You can pitch in by suggesting bug fixes and corrections as well as by expanding the code that's on offer. GitHub enables you to fork repositories and grow them with your own tweaks and features, and share those back to the main repository. If you come up with a new idea or approach, let me know. My team and I are happy to include great suggestions both at the repository and in the next edition of this Cookbook.

Getting Git

You can download this Cookbook's source code using the git version control system. An OS X implementation of git is available at http://code.google.com/p/git-osx-installer. OS X git implementations include both command-line and GUI solutions, so hunt around for the version that best suits your development needs.

Getting GitHub

GitHub (http://github.com) is the largest git-hosting site, with more than 150,000 public repositories. It provides both free hosting for public projects and paid options for private projects. With a custom web interface that includes wiki hosting, issue tracking, and an emphasis on social networking of project developers, it's a great place to find new code or collaborate on existing libraries. You can sign up for a free account at its Web site, enabling you to copy and modify the Cookbook repository or create your own open-source iOS projects to share with others.

Contacting the Author

If you have any comments or questions about this book, please drop me an e-mail message at erica@ericasadun.com, or stop by the github repository and contact me there.

Endnotes

1. No trouts, real or imaginary, were hurt in the development and production of this book. The same cannot be said for countless cans of Diet Coke that selflessly surrendered their contents in the service of this manuscript.
2. See the *Cocoa Fundamentals Guide* (http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf) for a head start on Cocoa, and for Xcode, see *A Tour of Xcode* (http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/A_Tour_of_Xcode/A_Tour_of_Xcode.pdf).
3. Big Nerd Ranch: www.bignerdranch.com.

Editor's Note: We Want to Hear from You!

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.

You can e-mail or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

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

E-mail: trina.macdonald@pearson.com

Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education, Inc.
75 Arlington St., Ste. 300
Boston, MA 02116

1

Device-Specific Development

Each iOS device represents a meld of unique, shared, momentary, and persistent properties. These properties include the device's current physical orientation, its model name, its battery state, and its access to onboard hardware. This chapter looks at the device from its build configuration to its active onboard sensors. It provides recipes that return a variety of information items about the unit in use. You read about testing for hardware prerequisites at runtime and specifying those prerequisites in the application's Info.plist file. You discover how to solicit sensor feedback via Core Motion and subscribe to notifications to create callbacks when sensor states change. You read about adding screen mirroring and second-screen output, and about soliciting device-specific details for tracking. This chapter covers the hardware, file system, and sensors available on the iPhone device and helps you programmatically take advantage of those features.

Accessing Basic Device Information

The `UIDevice` class exposes key device-specific properties, including the iPhone, iPad, or iPod touch model being used, the device name, and the OS name and version. It's a one-stop solution for pulling out certain system details. Each method is an instance method, which is called using the `UIDevice` singleton, via `[UIDevice currentDevice]`.

The system information you can retrieve from `UIDevice` includes these items:

- **`systemName`**—This returns the name of the operating system currently in use. For current generations of iOS devices, there is only one OS that runs on the platform: iPhone OS. Apple has not yet updated this name to match the general iOS rebranding.
- **`systemVersion`**—This value lists the firmware version currently installed on the unit: for example, 4.3, 5.1.1, 6.0, and so on.

- **`model`**—The iPhone model returns a string that describes its platform—namely iPhone, iPad, and iPod touch. Should iOS be extended to new devices, additional strings will describe those models. `localizedModel` provides a localized version of this property.
- **`userInterfaceIdiom`**—This property represents the interface style used on the current device, namely either iPhone (for iPhone and iPod touch) or iPad. Other idioms may be introduced as Apple offers additional platform styles.
- **`name`**—This string presents the iPhone name assigned by the user in iTunes, such as "Joe's iPhone" or "Binky." This name is also used to create the local hostname for the device.

Here are a few examples of these properties in use:

```
UIDevice *device = [UIDevice currentDevice];
NSLog(@"System name: %@", device.systemName);
NSLog(@"Model: %@", device.model);
NSLog(@"Name: %@", device.name);
```

For current iOS releases, you can use the idiom check with a simple Boolean test. Here's an example of how you might implement an iPad check. Notice the convenience macro. It tests for selector conformance and then returns `[UIDevice currentDevice].userInterfaceIdiom` if possible, and `UIUserInterfaceIdiomPhone` otherwise:

```
#define IS_IPAD (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)
```

Should this test fail, you may currently assume that you're working with an iPhone/iPod touch. If and when Apple releases a new family of devices, you'll need to update your code accordingly for a more nuanced test.

Adding Device Capability Restrictions

An application's Info.plist property list enables you to specify application requirements when you submit applications to iTunes. These restrictions enable you to tell iTunes what device features your application needs.

Each iOS unit provides a unique feature set. Some devices offer cameras and GPS capabilities. Others don't. Some have onboard gyros, autofocus, and other powerful options. You specify what features are needed to run your application on a device.

When you include the `UIRequiredDeviceCapabilities` key in your Info.plist file, iTunes limits application installation to devices that offer the required capabilities. Provide this list as either an array of strings or a dictionary.

An array specifies each required capability; each item in that array must be present on your device. A dictionary enables you to explicitly require or prohibit a feature. The dictionary keys are the capabilities. The dictionary values set whether the feature must be present (Boolean true) or omitted (Boolean false).

The current keys are detailed in Table 1-1. Only include those features that your application absolutely requires or cannot support. If your application can provide workarounds, do not add restrictions in this way. Table 1-1 discusses each feature in a positive sense. When using a prohibition rather than a requirement, reverse the meaning—for example, that an autofocus camera or gyro cannot be onboard, or that Game Center access cannot be supported.

Table 1-1 **Required Device Capabilities**

Key	Use
telephony	Application requires the Phone application or uses tel:// URLs.
wifi	Application requires local 802.11-based network access. If iOS must maintain that Wi-Fi connection as the app runs, add `UIRequiresPersistentWiFi` as a top-level property list key.
sms	Application requires the Messages application or uses sms:// URLs.
still-camera	Application requires an onboard still camera and can use the image picker interface to capture photos from that still camera.
auto-focus-camera	Application requires extra focus capabilities for macro photography or especially sharp images for in-image data detection.
front-facing-camera	Application requires a front-facing camera on the device.
camera-flash	Application requires a camera flash feature.
video-camera	Application requires a video-capable camera.
accelerometer	Application requires accelerometer-specific feedback beyond simple `UIViewController` orientation events.
gyroscope	Application requires an onboard gyroscope on the device.
location-services	Application uses Core Location of any kind.
gps	Application uses Core Location and requires the additional accuracy of GPS positioning.
magnetometer	Application uses Core Location and requires heading-related events—that is, the direction of travel. (The magnetometer is the built-in compass.)
gamekit	Application requires Game Center access (iOS 4.1 and later).
microphone	Application uses either built-in microphones or (approved) accessories that provide a microphone.
opengles-1	Application requires OpenGL ES 1.1.
opengles-2	Application requires OpenGL ES 2.0.
armv6	Application is compiled only for the armv6 instruction set (3.1 or later).
armv7	Application is compiled only for the armv7 instruction set (3.1 or later).

Key	Use
peer-peer	Application uses GameKit peer-to-peer connectivity over Bluetooth (3.1 or later).
bluetooth-le	Application requires Bluetooth low-energy support (5.0 and later).

For example, consider an application that offers an option for taking pictures when run on a camera-ready device. If the application otherwise works on pre-camera iPod touch units, do not include the still-camera restriction. Instead, use check for camera capability from within the application and present the camera option when appropriate. Adding a still-camera restriction eliminates many early iPod touch (first through third generation) and iPad (first generation) owners from your potential customer pool.

User Permission Descriptions

To protect privacy, the end user must explicitly permit your applications to access reminders, photos, location, contacts, and calendar data. To convince the user to opt-in, it helps to explain how your application can use this data and describe your reason for accessing it. Assign string values to the following keys at the top level of your Info.plist file. When iOS prompts your user for resource-specific permission, it displays these strings as part of its standard dialog box:

- `NSRemindersUsageDescription`
- `NSPhotoLibraryUsageDescription`
- `NSLocationUsageDescription`
- `NSContactsUsageDescription`
- `NSCalendarsUsageDescription`

Other Common Info.plist Keys

Here are a few other common keys you may want to assign in your property list, along with descriptions of what they do:

- **`UIFileSharingEnabled` (Boolean, defaults to off)**—Enables users to access the contents of your app's Documents folder from iTunes. This folder appears at the top level of your app sandbox.
- **`UIAppFonts` (Array, strings of font names including their extension)**—Specifies custom TTF fonts that you supply in your bundle. When added, you access them using standard `UIFont` calls.
- **`UIApplicationExitsOnSuspend` (Boolean, defaults to off)**—Enables your app to terminate when the user clicks the Home button rather than move to the background. When enabled, iOS terminates the app and purges it from memory.

- **`UIRequiresPersistentWifi` (Boolean, defaults to off)**—Instructs iOS to maintain a Wi-Fi connection while the app is active.
- **`UIStatusBarHidden` (Boolean, defaults to off)**—If enabled, hides the status bar as the app launches.
- **`UIStatusBarStyle` (String, defaults to UIStatusBarStyleDefault)**—Specifies the style of the status bar at app launch.

Recipe: Checking Device Proximity and Battery States

The `UIDevice` class offers APIs that enable you to keep track of device characteristics including the states of the battery and proximity sensor. Recipe 1-1 demonstrates how you can enable and query monitoring for these two technologies. Both provide updates in the form of notifications, which you can subscribe to so your application is informed of important updates.

Enabling and Disabling the Proximity Sensor

Proximity is an iPhone-specific feature at this time. The iPod touch and iPad do not offer proximity sensors. Unless you have some pressing reason to hold an iPhone against body parts (or vice versa), using the proximity sensor accomplishes little.

When enabled, it has one primary task. It detects whether there's a large object right in front of it. If so, it switches the screen off and sends a general notification. Move the blocking object away and the screen switches back on. This prevents you from pressing buttons or dialing the phone with your ear when you are on a call. Some poorly designed protective cases keep the iPhone's proximity sensors from working properly.

Siri uses this feature. When you hold the phone up to your ear, it records your query, sending it to be interpreted. Siri's voice interface does not depend on a visual GUI to operate.

Recipe 1-1 also demonstrates how to work with proximity sensing on the iPhone. Its code uses the `UIDevice` class to toggle proximity monitoring and subscribes to `UIDeviceProximity StateDidChangeNotification` to catch state changes. The two states are on and off. When the `UIDevice proximityState` property returns `YES`, the proximity sensor has been activated.

Monitoring the Battery State

You can programmatically keep track of the battery and device state. These APIs enable you to know the level to which the battery is charged and whether the device is plugged into a charging source. The battery level is a floating-point value that ranges between 1.0 (fully charged) and 0.0 (fully discharged). It provides an approximate discharge level that you can use to query before performing operations that put unusual strain on the device.

For example, you might want to caution your user about performing a large series of mathematical computations and suggest that the user plug in to a power source. You retrieve the battery level via this `UIDevice` call. The value returned is produced in 5% increments:

```
NSLog(@"Battery level: %0.2f%",
    [UIDevice currentDevice].batteryLevel * 100);
```

The charge state has four possible values. The unit can be charging (that is, connected to a power source), full, unplugged, and a catchall "unknown." Recover the state using the UIDevice batteryState property:

```
NSArray *stateArray = @[
    @"Battery state is unknown",
    @"Battery is not plugged into a charging source",
    @"Battery is charging",
    @"Battery state is full"];

NSLog(@"Battery state: %@",
    stateArray[[UIDevice currentDevice].batteryState]);
```

Don't think of these choices as persistent states. Instead, think of them as momentary reflections of what is actually happening to the device. They are not flags. They are not OR'ed together to form a general battery description. Instead, these values reflect the most recent state change.

You can easily monitor state changes by responding to notifications that the battery state has changed. In this way, you can catch momentary events, such as when the battery finally recharges fully, when the user has plugged in to a power source to recharge, and when the user disconnects from that power source.

To start monitoring, set the `batteryMonitoringEnabled` property to `YES`. During monitoring, the `UIDevice` class produces notifications when the battery state or level changes. Recipe 1-1 subscribes to both notifications. Please note that you can also check these values directly, without waiting for notifications. Apple provides no guarantees about the frequency of level change updates, but as you can tell by testing this recipe, they arrive in a fairly regular fashion.

Recipe 1-1 **Monitoring Proximity and Battery**

```
// View the current battery level and state
- (void) peekAtBatteryState
{
    NSArray *stateArray = [NSArray arrayWithObjects:
                           @"Battery state is unknown",
                           @"Battery is not plugged into a charging source",
                           @"Battery is charging",
                           @"Battery state is full", nil];

    NSString *status = [NSString stringWithFormat:
```

```
        @"Battery state: %@, Battery level: %0.2f%%",
        [stateArray objectAtIndex:[UIDevice currentDevice].batteryState],
        [UIDevice currentDevice].batteryLevel * 100];

    NSLog(@"%@", status);
}

// Show whether proximity is being monitored
- (void) updateTitle
{
    self.title = [NSString stringWithFormat:@"Proximity %@",
       [UIDevice currentDevice].proximityMonitoringEnabled ? @"On" : @"Off"];
}

// Toggle proximity monitoring off and on
- (void) toggle: (id) sender
{
    // Determine the current proximity monitoring and toggle it
    BOOL isEnabled = [UIDevice currentDevice].proximityMonitoringEnabled;
    [UIDevice currentDevice].proximityMonitoringEnabled = !isEnabled;
    [self updateTitle];
}

- (void) loadView
{
    [super loadView];

    // Enable toggling and initialize title
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Toggle", @selector(toggle:));
    [self updateTitle];

    // Add proximity state checker
    [[NSNotificationCenter defaultCenter]
        addObserverForName:UIDeviceProximityStateDidChangeNotification
        object:nil queue:[NSOperationQueue mainQueue]
        usingBlock:^(NSNotification *notification) {
            // Sensor has triggered either on or off
            NSLog(@"The proximity sensor %@",
                [UIDevice currentDevice].proximityState ?
                @"will now blank the screen" : @"will now restore the screen");
    }];

    // Enable battery monitoring
    [[UIDevice currentDevice] setBatteryMonitoringEnabled:YES];
```

```
    // Add observers for battery state and level changes
    [[NSNotificationCenter defaultCenter]
        addObserverForName:UIDeviceBatteryStateDidChangeNotification
        object:nil queue:[NSOperationQueue mainQueue]
        usingBlock:^(NSNotification *notification) {
            // State has changed
            NSLog(@"Battery State Change");
            [self peekAtBatteryState];
    }];

    [[NSNotificationCenter defaultCenter]
        addObserverForName:UIDeviceBatteryLevelDidChangeNotification
        object:nil queue:[NSOperationQueue mainQueue]
        usingBlock:^(NSNotification *notification) {
            // Level has changed
            NSLog(@"Battery Level Change");
            [self peekAtBatteryState];
    }];
}
```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Advanced-Cookbook and go to the folder for Chapter 1.

Detecting Retina Support

In recent years, Apple introduced the Retina display on its flagship devices. Its pixel density is, according to Apple, high enough so the human eye cannot distinguish individual pixels. Apps shipped with higher-resolution art take advantage of this improved display quality.

The `UIScreen` class offers an easy way to check whether the current device offers a built-in Retina display. Check the screen `scale` property, which provides the factor that converts from the logical coordinate space (points, approximately $1/160^{th}$ of an inch) into a device coordinate space (pixels). It is 1.0 for standard displays, so one point corresponds to one pixel. It is 2.0 for Retina displays (4 pixels per point):

```
- (BOOL) hasRetinaDisplay
{
    return ([UIScreen mainScreen].scale == 2.0f);
}
```

The `UIScreen` class also offers two useful display-size properties. The `bounds` returns the screen's bounding rectangle, measured in points. This gives you the full size of the screen, regardless of any onscreen elements such as status bars, navigation bars, or tab bars. The

`applicationFrame` property, also measured in points, excludes the status bar, providing the frame for your application's initial window size.

Recipe: Recovering Additional Device Information

Both `sysctl()` and `sysctlbyname()` enable you to retrieve system information. These standard UNIX functions query the operating system about hardware and OS details. You can get a sense of the kind of scope on offer by glancing at the /usr/include/sys/sysctl.h include file on the Macintosh. There you can find an exhaustive list of constants that can be used as parameters to these functions.

These constants enable you to check for core information such as the system's CPU frequency, the amount of available memory, and more. Recipe 1-2 demonstrates this functionality. It introduces a `UIDevice` category that gathers system information and returns it via a series of method calls.

You might wonder why this category includes a platform method, when the standard `UIDevice` class returns device models on demand. The answer lies in distinguishing different types of units.

An iPhone 3GS's model is simply "iPhone," as is the model of an iPhone 4S. In contrast, this recipe returns a platform value of "iPhone2,1" for the 3GS and "iPhone 4,1" for the iPhone 4S. This enables you to programmatically differentiate the 3GS unit from a first-generation iPhone ("iPhone1,1") or iPhone 3G ("iPhone1,2").

Each model offers distinct built-in capabilities. Knowing exactly which iPhone you're dealing with helps you determine whether that unit likely supports features such as accessibility, GPS, and magnetometers.

Recipe 1-2 **Extending Device Information Gathering**

```
@implementation UIDevice (Hardware)
+ (NSString *) getSysInfoByName:(char *)typeSpecifier
{
    // Recover sysctl information by name
    size_t size;
    sysctlbyname(typeSpecifier, NULL, &size, NULL, 0);

    char *answer = malloc(size);
    sysctlbyname(typeSpecifier, answer, &size, NULL, 0);

    NSString *results = [NSString stringWithCString:answer
        encoding: NSUTF8StringEncoding];
    free(answer);

    return results;
}
```

```
- (NSString *) platform
{
    return [UIDevice getSysInfoByName:"hw.machine"];
}

- (NSUInteger) getSysInfo: (uint) typeSpecifier
{
    size_t size = sizeof(int);
    int results;
    int mib[2] = {CTL_HW, typeSpecifier};
    sysctl(mib, 2, &results, &size, NULL, 0);
    return (NSUInteger) results;
}

- (NSUInteger) cpuFrequency
{
    return [UIDevice getSysInfo:HW_CPU_FREQ];
}

- (NSUInteger) busFrequency
{
    return [UIDevice getSysInfo:HW_BUS_FREQ];
}

- (NSUInteger) totalMemory
{
    return [UIDevice getSysInfo:HW_PHYSMEM];
}

- (NSUInteger) userMemory
{
    return [UIDevice getSysInfo:HW_USERMEM];
}

- (NSUInteger) maxSocketBufferSize
{
    return [UIDevice getSysInfo:KIPC_MAXSOCKBUF];
}
@end
```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Advanced-Cookbook and go to the folder for Chapter 1.

Recipe: Using Acceleration to Locate “Up”

The iPhone provides three onboard sensors that measure acceleration along the iPhone’s perpendicular axes: left/right (`x`), up/down (`y`), and front/back (`z`). These values indicate the forces affecting the iPhone, from both gravity and user movement. You can get some neat force feedback by swinging the iPhone around your head (centripetal force) or dropping it from a tall building (freefall). Unfortunately, you might not recover that data after your iPhone becomes an expensive bit of scrap metal.

To subscribe an object to iPhone accelerometer updates, set it as the delegate. The object set as the delegate must implement the `UIAccelerometerDelegate` protocol:

```
[[UIAccelerometer sharedAccelerometer] setDelegate:self]
```

When assigned, your delegate receives accelerometer:didAccelerate: callback messages, which you can track and respond to. The UIAcceleration structure sent to the delegate method consists of floating-point values for the x, y, and z axes. Each value ranges from –1.0 to 1.0:

```
float x = acceleration.x;
float y = acceleration.y;
float z = acceleration.z;
```

Recipe 1-3 uses these values to help determine the “up” direction. It calculates the arctangent between the X and Y acceleration vectors, returning the up-offset angle. As new acceleration messages are received, the recipe rotates a `UIImageView` instance with its picture of an arrow, which you can see in Figure 1-1, to point up. The real-time response to user actions ensures that the arrow continues pointing upward, no matter how the user reorients the phone.

Recipe 1-3 **Catching Acceleration Events**

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    // Determine up from the x and y acceleration components
    float xx = -acceleration.x;
    float yy = acceleration.y;
    float angle = atan2(yy, xx);
    [arrow setTransform:
        CGAffineTransformMakeRotation(angle)];
}


- (void) viewDidLoad
{
    // Initialize the delegate to start catching accelerometer events
    [UIAccelerometer sharedAccelerometer].delegate = self;
}
```

Figure 1-1 A little math recovers the "up" direction by performing an arctan function using the x and y force vectors. In this example, the arrow always points up, no matter how the user reorients the iPhone.

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Advanced-Cookbook and go to the folder for Chapter 1.

Working with Basic Orientation

The `UIDevice` class uses the built-in `orientation` property to retrieve the physical orientation of the device. iOS devices support seven possible values for this property:

- **`UIDeviceOrientationUnknown`**—The orientation is currently unknown.
- **`UIDeviceOrientationPortrait`**—The home button is down.
- **`UIDeviceOrientationPortraitUpsideDown`**—The home button is up.
- **`UIDeviceOrientationLandscapeLeft`**—The home button is to the right.
- **`UIDeviceOrientationLandscapeRight`**—The home button is to the left.

- **`UIDeviceOrientationFaceUp`**—The screen is face up.
- **`UIDeviceOrientationFaceDown`**—The screen is face down.

The device can pass through any or all of these orientations during a typical application session. Although orientation is created in concert with the onboard accelerometer, these orientations are not tied in any way to a built-in angular value.

iOS offers two built-in macros to help determine if a device orientation enumerated value is portrait or landscape: namely `UIDeviceOrientationIsPortrait()` and `UIDeviceOrientationIsLandscape()`. It is convenient to extend the `UIDevice` class to offer these tests as built-in device properties:

```
@property (nonatomic, readonly) BOOL isLandscape;
@property (nonatomic, readonly) BOOL isPortrait;

- (BOOL) isLandscape
{
    return UIDeviceOrientationIsLandscape(self.orientation);
}

- (BOOL) isPortrait
{
    return UIDeviceOrientationIsPortrait(self.orientation);
}
```

Your code can subscribe directly to device reorientation notifications. To accomplish this, send `beginGeneratingDeviceOrientationNotifications` to the `currentDevice` singleton. Then add an observer to catch the ensuing `UIDeviceOrientationDidChangeNotification` updates. As you would expect, you can finish listening by calling `endGeneratingDeviceOrientationNotification`.

Retrieving the Current Accelerometer Angle Synchronously

At times you may want to query the accelerometer without setting yourself up as a full delegate. The following methods, which are meant for use within a `UIDevice` category, enable you to synchronously return the current device angle along the x/y plane—the front face plane of the iOS device. Accomplish this by entering a new run loop, wait for an accelerometer event, retrieve the current angle from that callback, and then leave the run loop to return that angle:

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    float xx = acceleration.x;
    float yy = -acceleration.y;
    device_angle = M_PI / 2.0f - atan2(yy, xx);
```

```
    if (device_angle > M_PI)
        device_angle -= 2 * M_PI;

    CFRunLoopStop(CFRunLoopGetCurrent());
}

- (float) orientationAngle
{
    // Supercede current delegate
    id priorDelegate = [UIAccelerometer sharedAccelerometer].delegate;
    [UIAccelerometer sharedAccelerometer].delegate = self;

    // Wait for a reading
    CFRunLoopRun();

    // Restore delegate
    [UIAccelerometer sharedAccelerometer].delegate = priorDelegate;

    return device_angle;
}
```

This is not an approach to use for continuous polling—use the callbacks directly for that. But for an occasional angle query, these methods provide simple and direct access to the current screen angle.

Calculating Orientation from the Accelerometer

The `UIDevice` class does not report a proper orientation when applications are first launched. It updates the orientation only after the device has moved into a new position or `UIViewController` methods kick in.

An application launched in portrait orientation may not read as "portrait" until the user moves the device out of and then back into the proper orientation. This condition exists on the simulator and on the iPhone device and is easily tested. (Radars for this issue have been closed with updates that the features are working as designed.)

For a workaround, consider recovering the angular orientation directly as just shown. Then, after you determine the device angle, convert from the accelerometer-based angle to a device orientation. Here's how that might work in code:

```
// Limited to the four portrait/landscape options
- (UIDeviceOrientation) acceleratorBasedOrientation
{
    CGFloat baseAngle = self.orientationAngle;
    if ((baseAngle > -M_PI_4) && (baseAngle < M_PI_4))
        return UIDeviceOrientationPortrait;
    if ((baseAngle < -M_PI_4) && (baseAngle > -3 * M_PI_4))
```

```
        return UIDeviceOrientationLandscapeLeft;
    if ((baseAngle > M_PI_4) && (baseAngle < 3 * M_PI_4))
        return UIDeviceOrientationLandscapeRight;
    return UIDeviceOrientationPortraitUpsideDown;
}
```

Be aware that this example looks only at the x-y plane, which is where most user interface decisions need to be made. This snippet completely ignores the z-axis, meaning that you'll end up with vaguely random results for the face-up and face-down orientations. Adapt this code to provide that nuance if needed.

The `UIViewController` class's `interfaceOrientation` instance method reports the orientation of a view controller's interface. Although this is not a substitute for accelerometer readings, many interface layout issues rest on the underlying view orientation rather than device characteristics.

Be aware that, especially on the iPad, a child view controller may use a layout orientation that's distinct from a device orientation. For example, an embedded controller may present a portrait layout within a landscape split view controller. Even so, consider whether your orientation-detection code is satisfiable by the underlying interface orientation. It may be more reliable than device orientation, especially as the application launches. Develop accordingly.

Calculate a Relative Angle

Screen reorientation support means that an interface's relationship to a given device angle must be supported in quarters, one for each possible front-facing screen orientation. As the `UIViewController` automatically rotates its onscreen view, the math needs to catch up to account for those reorientations.

The following method, which is written for use in a `UIDevice` category, calculates angles so that the angle remains in synchrony with the device orientation. This creates simple offsets from vertical that match the way the GUI is currently presented:

```
- (float) orientationAngleRelativeToOrientation:
    (UIDeviceOrientation) someOrientation
{
     float dOrientation = 0.0f;
    switch (someOrientation)
    {
        case UIDeviceOrientationPortraitUpsideDown:
            {dOrientation = M_PI; break;}
        case UIDeviceOrientationLandscapeLeft:
            {dOrientation = -(M_PI/2.0f); break;}
        case UIDeviceOrientationLandscapeRight:
            {dOrientation = (M_PI/2.0f); break;}
        default: break;
    }
```

```
    float adjustedAngle =
        fmod(self.orientationAngle - dOrientation, 2.0f * M_PI);
    if (adjustedAngle > (M_PI + 0.01f))
        adjustedAngle = (adjustedAngle - 2.0f * M_PI);
    return adjustedAngle;
}
```

This method uses a floating-point modulo to retrieve the difference between the actual screen angle and the interface orientation angular offset to return that all-important vertical angular offset.

> **Note**
>
> In iOS 6, use your Info.plist to allow and disallow orientation changes instead of `shouldAutorotateToInterfaceOrientation:`.

Recipe: Using Acceleration to Move Onscreen Objects

With a bit of programming, the iPhone's onboard accelerometer can make objects "move" around the screen, responding in real time to the way the user tilts the phone. Recipe 1-4 builds an animated butterfly that users can slide across the screen.

The secret to make this work lies in adding what a "physics timer" to the program. Instead of responding directly to changes in acceleration, the way Recipe 1-3 did, the accelerometer callback measures the current forces. It's up to the timer routine to apply those forces to the butterfly over time by changing its frame. Here are some key points to keep in mind:

- As long as the direction of force remains the same, the butterfly accelerates. Its velocity increases, scaled according to the degree of acceleration force in the X or Y direction.
- The `tick` routine, called by the timer, moves the butterfly by adding the velocity vector to the butterfly's origin.
- The butterfly's range is bounded. So when it hits an edge, it stops moving in that direction. This keeps the butterfly onscreen at all times. The `tick` method checks for boundary conditions. For example, if the butterfly hits a vertical edge, it can still move horizontally.
- The butterfly reorients itself so it always falling "down." This happens by applying a simple rotation transform in the `tick` method. Be careful when using transforms in addition to frame or center offsets. Always reset the math before applying offsets, and then reapply any angular changes. Failing to do so may cause your frames to zoom, shrink, or skew unexpectedly.

> **Note**
>
> Timers in their natural state do not work with blocks. If you'd rather use a block-based design, check around github to find workarounds that do.

Recipe 1-4 **Sliding an Onscreen Object Based on Accelerometer Feedback**

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    // Extract the acceleration components
    float xx = -acceleration.x;
    float yy = acceleration.y;

    // Store the most recent angular offset
    mostRecentAngle = atan2(yy, xx);

    // Has the direction changed?
    float accelDirX = SIGN(xvelocity) * -1.0f;
    float newDirX = SIGN(xx);
    float accelDirY = SIGN(yvelocity) * -1.0f;
    float newDirY = SIGN(yy);

    // Accelerate. To increase viscosity lower the additive value
    if (accelDirX == newDirX) xaccel =
        (abs(xaccel) + 0.85f) * SIGN(xaccel);
    if (accelDirY == newDirY) yaccel =
        (abs(yaccel) + 0.85f) * SIGN(yaccel);

    // Apply acceleration changes to the current velocity
    xvelocity = -xaccel * xx;
    yvelocity = -yaccel * yy;
}

- (void) tick
{
    // Reset the transform before changing position
    butterfly.transform = CGAffineTransformIdentity;

    // Move the butterfly according to the current velocity vector
    CGRect rect = CGRectOffset(butterfly.frame, xvelocity, 0.0f);
    if (CGRectContainsRect(self.view.bounds, rect))
        butterfly.frame = rect;
```

```
    rect = CGRectOffset(butterfly.frame, 0.0f, yvelocity);
    if (CGRectContainsRect(self.view.bounds, rect))
        butterfly.frame = rect;

    // Rotate the butterfly independently of position
    butterfly.transform =
        CGAffineTransformMakeRotation(mostRecentAngle + M_PI_2);
}

- (void) initButterfly
{
    CGSize size;

    // Load the animation cells
    NSMutableArray *butterflies = [NSMutableArray array];
    for (int i = 1; i <= 17; i++)
    {
        NSString *fileName = [NSString stringWithFormat:@"bf_%d.png", i];
        UIImage *image = [UIImage imageNamed:fileName];
        size = image.size;
        [butterflies addObject:image];
    }

    // Begin the animation
    butterfly = [[UIImageView alloc]
        initWithFrame:(CGRect){.size=size}];
    [butterfly setAnimationImages:butterflies];
    butterfly.animationDuration = 0.75f;
    [butterfly startAnimating];

    // Set the butterfly's initial speed and acceleration
    xaccel = 2.0f;
    yaccel = 2.0f;
    xvelocity = 0.0f;
    yvelocity = 0.0f;

    // Add the butterfly
    butterfly.center = RECTCENTER(self.view.bounds);
    [self.view addSubview:butterfly];


    // Activate the accelerometer
    [[UIAccelerometer sharedAccelerometer] setDelegate:self];

    // Start the physics timer
    [NSTimer scheduledTimerWithTimeInterval: 0.03f
```

```
        target: self selector: @selector(tick)
        userInfo: nil repeats: YES];
}
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Advanced-Cookbook and go to the folder for Chapter 1.

Recipe: Accelerometer-Based Scroll View

Several readers asked me to include a tilt scroller recipe in this edition. A tilt scroller uses the device's built-in accelerometer to control movement around a `UIScrollView`'s content. As the user adjusts the device, the material "falls down" accordingly. Instead of a view being positioned onscreen, the content view scrolls to a new offset.

The challenge in creating this interface lies in determining where the device should have its resting axis. Most people would initially suggest that the display should stabilize when lying on its back, with the Z-direction pointed straight up in the air. It turns out that's actually a fairly bad design choice. To use that axis means the screen must actually tilt away from the viewer during navigation. With the device rotated away from view, the user cannot fully see what is happening onscreen, especially when using the device in a seated position and somewhat when looking at the device while standing overhead.

Instead, Recipe 1-5 assumes that the stable position is created by the Z-axis pointing at approximately 45 degrees, the natural position users holding an iPhone or iPad in their hands. This is halfway between a face-up and a face-forward position. The math in Recipe 1-5 is adjusted accordingly. Tilting back and forward from this slanting position leaves the screen with maximal visibility during adjustments.

The other change in this recipe, compared to Recipe 1-4, is the much lower acceleration constant. This enables onscreen movement to happen more slowly, letting users more easily slow down and resume navigation.

Recipe 1-5 **Tilt Scroller**

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    // extract the acceleration components
    float xx = -acceleration.x;
    float yy = (acceleration.z + 0.5f) * 2.0f; // between face-up and face-forward

    // Has the direction changed?
    float accelDirX = SIGN(xvelocity) * -1.0f;
```

```
    float newDirX = SIGN(xx);
    float accelDirY = SIGN(yvelocity) * -1.0f;
    float newDirY = SIGN(yy);

    // Accelerate. To increase viscosity lower the additive value
    if (accelDirX == newDirX) xaccel = (abs(xaccel) + 0.005f) * SIGN(xaccel);
    if (accelDirY == newDirY) yaccel = (abs(yaccel) + 0.005f) * SIGN(yaccel);

    // Apply acceleration changes to the current velocity
    xvelocity = -xaccel * xx;
    yvelocity = -yaccel * yy;
}

- (void) tick
{
    xoff += xvelocity;
    xoff = MIN(xoff, 1.0f);
    xoff = MAX(xoff, 0.0f);

    yoff += yvelocity;
    yoff = MIN(yoff, 1.0f);
    yoff = MAX(yoff, 0.0f);

    // update the content offset based on the current velocities
    CGFloat xsize = sv.contentSize.width - sv.frame.size.width;
    CGFloat ysize = sv.contentSize.height - sv.frame.size.height;
    sv.contentOffset = CGPointMake(xoff * xsize, yoff * ysize);
}

- (void) viewDidAppear:(BOOL)animated
{
    NSString *map = @"http://maps.weather.com/images/\
        maps/current/curwx_720x486.jpg";
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [queue addOperationWithBlock:
     ^{
         // Load the weather data
         NSURL *weatherURL = [NSURL URLWithString:map];
         NSData *imageData = [NSData dataWithContentsOfURL:weatherURL];

         // Update the image on the main thread using the main queue
         [[NSOperationQueue mainQueue] addOperationWithBlock:^{
             UIImage *weatherImage = [UIImage imageWithData:imageData];
             UIImageView *imageView =
                 [[UIImageView alloc] initWithImage:weatherImage];
             CGSize initSize = weatherImage.size;
             CGSize destSize = weatherImage.size;
```

```
            // Ensure that the content size is significantly bigger
            // than the screen can show at once
            while ((destSize.width < (self.view.frame.size.width * 4)) ||
                   (destSize.height < (self.view.frame.size.height * 4)))
            {
                destSize.width += initSize.width;
                destSize.height += initSize.height;
            }

            imageView.userInteractionEnabled = NO;
            imageView.frame = (CGRect){.size = destSize};
            sv.contentSize = destSize;

            [sv addSubview:imageView];

            // Activate the accelerometer
            [[UIAccelerometer sharedAccelerometer] setDelegate:self];

            // Start the physics timer
            [NSTimer scheduledTimerWithTimeInterval: 0.03f
                target: self selector: @selector(tick)
                userInfo: nil repeats: YES];
        }];
    }];
}
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Advanced-Cookbook and go to the folder for Chapter 1.

Recipe: Core Motion Basics

The Core Motion framework centralizes motion data processing. Introduced in the iOS 4 SDK, Core Motion supersedes the direct accelerometer access you've just read about. It provides centralized monitoring of three key onboard sensors. These sensors are composed of the gyroscope, which measures device rotation; the magnetometer, which provides a way to measure compass bearings; and the accelerometer, which detects gravitational changes along three axes. A fourth entry point called *device motion* combines all three of these sensors into a single monitoring system.

Core Motion uses raw values from these sensors to create readable measurements, primarily in the form of force vectors. Measurable items include the following properties:

- **Device attitude (`attitude`)**—The device's orientation relative to some frame of reference. The attitude is represented as a triplet of roll, pitch, and yaw angles, each measured in radians.
- **Rotation rate (`rotationRate`)**—The rate at which the device rotates around each of its three axes. The rotation includes x, y, and z angular velocity values measured in radians per second.
- **Gravity (`gravity`)**—The device's current acceleration vector as imparted by the normal gravitational field. Gravity is measured in g's, along the x, y, and z axes. Each unit represents the standard gravitational force imparted by Earth (namely 32 feet per second per second, or 9.8 meters per second per second).
- **User acceleration (`userAcceleration`)**—The acceleration vector being imparted by the user. Like `gravity`, user acceleration is measured in g's along the x, y, and z axes. When added together, the user vector and the gravity vector represent the total acceleration imparted to the device.
- **Magnetic field (`magneticField`)**—The vector representing the overall magnetic field values in the device's vicinity. The field is measured in microteslas along the x, y, and z axes. A calibration accuracy is also provided, to inform your application of the field measurements quality.

Testing for Sensors

As you read earlier in this chapter, you can use the application's Info.plist file to require or exclude onboard sensors. You can also test an in-app for each kind of possible Core Motion support:

```
if (motionManager.gyroAvailable)
    [motionManager startGyroUpdates];

if (motionManager.magnetometerAvailable)
    [motionManager startMagnetometerUpdates];

if (motionManager.accelerometerAvailable)
    [motionManager startAccelerometerUpdates];

if (motionManager.deviceMotionAvailable)
    [motionManager startDeviceMotionUpdates];
```

Starting updates does not produce a delegate callback mechanism like you encountered with the `UIAccelerometer` class. Instead, you are responsible for polling each value, or you can use a block-based update mechanism that executes a block that you provide at each update (for example, `startAccelerometerUpdatesToQueue:withHandler:`).

Handler Blocks

Recipe 1-6 adapts Recipe 1-4 for use with Core Motion. The acceleration callback has been moved into a handler block, and the x and y values are read from the data's acceleration property. Otherwise, the code remains unchanged. Here, you see the Core Motion basics: A new motion manager is created. It tests for accelerometer availability. It then starts updates using a new operation queue, which persists for the duration of the application run.

The `establishMotionManager` and `shutDownMotionManager` methods enable your application to start up and shut down the motion manager on demand. These methods are called from the application delegate when the application becomes active and when it suspends:

```
- (void) applicationWillResignActive:(UIApplication *)application
{
    [tbvc shutDownMotionManager];
}

- (void) applicationDidBecomeActive:(UIApplication *)application
{
    [tbvc establishMotionManager];
}
```

These methods provide a clean way to shut down and resume motion services in response to the current application state.

Recipe 1-6 **Basic Core Motion**

```
@implementation TestBedViewController
- (void) tick
{
    butterfly.transform = CGAffineTransformIdentity;

    // Move the butterfly according to the current velocity vector
    CGRect rect = CGRectOffset(butterfly.frame, xvelocity, 0.0f);
    if (CGRectContainsRect(self.view.bounds, rect))
        butterfly.frame = rect;

    rect = CGRectOffset(butterfly.frame, 0.0f, yvelocity);
    if (CGRectContainsRect(self.view.bounds, rect))
        butterfly.frame = rect;

    butterfly.transform =
       CGAffineTransformMakeRotation(mostRecentAngle + M_PI_2);
}

- (void) shutDownMotionManager
{
    NSLog(@"Shutting down motion manager");
```

```
    [motionManager stopAccelerometerUpdates];
    motionManager = nil;

    [timer invalidate];
    timer = nil;
}

- (void) establishMotionManager
{
    if (motionManager)
        [self shutDownMotionManager];

    NSLog(@"Establishing motion manager");

    // Establish the motion manager
    motionManager = [[CMMotionManager alloc] init];
    if (motionManager.accelerometerAvailable)
        [motionManager
         startAccelerometerUpdatesToQueue:
             [[NSOperationQueue alloc] init]
         withHandler:^(CMAccelerometerData *data, NSError *error)
         {
             // Extract the acceleration components
             float xx = -data.acceleration.x;
             float yy = data.acceleration.y;
             mostRecentAngle = atan2(yy, xx);

             // Has the direction changed?
             float accelDirX = SIGN(xvelocity) * -1.0f;
             float newDirX = SIGN(xx);
             float accelDirY = SIGN(yvelocity) * -1.0f;
             float newDirY = SIGN(yy);

             // Accelerate. To increase viscosity,
             // lower the additive value
             if (accelDirX == newDirX)
                 xaccel = (abs(xaccel) + 0.85f) * SIGN(xaccel);
             if (accelDirY == newDirY)
                 yaccel = (abs(yaccel) + 0.85f) * SIGN(yaccel);

             // Apply acceleration changes to the current velocity
             xvelocity = -xaccel * xx;
             yvelocity = -yaccel * yy;
         }];

    // Start the physics timer
    timer = [NSTimer scheduledTimerWithTimeInterval: 0.03f
```

```
        target: self selector: @selector(tick)
        userInfo: nil repeats: YES];
}

- (void) initButterfly
{
    CGSize size;

    // Load the animation cells
    NSMutableArray *butterflies = [NSMutableArray array];
    for (int i = 1; i <= 17; i++)
    {
        NSString *fileName =
            [NSString stringWithFormat:@"bf_%d.png", i];
        UIImage *image = [UIImage imageNamed:fileName];
        size = image.size;
        [butterflies addObject:image];
    }

    // Begin the animation
    butterfly = [[UIImageView alloc]
        initWithFrame:(CGRect){.size=size}];
    [butterfly setAnimationImages:butterflies];
    butterfly.animationDuration = 0.75f;
    [butterfly startAnimating];

    // Set the butterfly's initial speed and acceleration
    xaccel = 2.0f;
    yaccel = 2.0f;
    xvelocity = 0.0f;
    yvelocity = 0.0f;

    // Add the butterfly
    butterfly.center = RECTCENTER(self.view.bounds);
    [self.view addSubview:butterfly];
}

- (void) loadView
{
    [super loadView];
    self.view.backgroundColor = [UIColor whiteColor];
    [self initButterfly];
}
@end
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Advanced-Cookbook and go to the folder for Chapter 1.

Recipe: Retrieving and Using Device Attitude

Imagine an iPad sitting on a desk. There's an image displayed on the iPad, which you can bend over and look at. Now imagine rotating that iPad as it lays flat on the desk, but as the iPad moves, the image does not. It maintains a perfect alignment with the world around it. Regardless of how you spin the iPad, the image doesn't "move" as the view updates to balance the physical movement. That's how Recipe 1-7 works, taking advantage of a device's onboard gyroscope—a necessary requirement to make this recipe work.

The image adjusts however you hold the device. In addition to that flat manipulation, you can pick up the device and orient it in space. If you flip the device and look at it over your head, you see the reversed "bottom" of the image. You can also tilt it along both axes: the one that runs from the home button to the camera, and the other that runs along the surface of the iPad, from the midpoints between the camera and home button. The other axis, the one you first explore, is coming out of the device from its middle, pointing to the air above the device and passing through that middle point to behind it. As you manipulate the device, the image responds to create a virtual still world within that iPad.

Recipe 1-7 shows how to do this with just a few simple geometric transformations. It establishes a motion manager, subscribes to device motion updates, and then applies image transforms based on the roll, pitch, and yaw returned by the motion manager.

Recipe 1-7 **Using Device Motion Updates to Fix an Image in Space**

```
- (void) shutDownMotionManager
{
    NSLog(@"Shutting down motion manager");
    [motionManager stopDeviceMotionUpdates];
    motionManager = nil;
}

- (void) establishMotionManager
{
    if (motionManager)
        [self shutDownMotionManager];

    NSLog(@"Establishing motion manager");

    // Establish the motion manager
```

```
    motionManager = [[CMMotionManager alloc] init];
    if (motionManager.deviceMotionAvailable)
        [motionManager
         startDeviceMotionUpdatesToQueue:
             [NSOperationQueue currentQueue]
         withHandler: ^(CMDeviceMotion *motion, NSError *error) {
             CATransform3D transform;
             transform = CATransform3DMakeRotation(
                 motion.attitude.pitch, 1, 0, 0);
             transform = CATransform3DRotate(transform,
                 motion.attitude.roll, 0, 1, 0);
             transform = CATransform3DRotate(transform,
                 motion.attitude.yaw, 0, 0, 1);
             imageView.layer.transform = transform;
     }];
}
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Advanced-Cookbook and go to the folder for Chapter 1.

Detecting Shakes Using Motion Events

When the iPhone detects a motion event, it passes that event to the current first responder, the primary object in the responder chain. Responders are objects that can handle events. All views and windows are responders and so is the application object.

The responder chain provides a hierarchy of objects, all of which can respond to events. When an object toward the start of the chain receives an event, that event does not get passed further down. The object handles it. If it cannot, that event can move on to the next responder.

Objects often become the first responder by declaring themselves to be so, via `becomeFirstResponder`. In this snippet, a `UIViewController` ensures that it becomes the first responder whenever its view appears onscreen. Upon disappearing, it resigns the first responder position:

```
- (BOOL)canBecomeFirstResponder {
    return YES;
}

// Become first responder whenever the view appears
- (void)viewDidAppear:(BOOL)animated {
```

```
    [super viewDidAppear:animated];
    [self becomeFirstResponder];
}

// Resign first responder whenever the view disappears
- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [self resignFirstResponder];
}
```

First responders receive all touch and motion events. The motion callbacks mirror `UIView` touch callback stages. The callback methods are as follows:

- **`motionBegan:withEvent:`**—This callback indicates the start of a motion event. At the time of writing this book, there was only one kind of motion event recognized: a shake. This may not hold true for the future, so you might want to check the motion type in your code.
- **`motionEnded:withEvent:`**—The first responder receives this callback at the end of the motion event.
- **`motionCancelled:withEvent:`**—As with touches, motions can be canceled by incoming phone calls and other system events. Apple recommends that you implement all three motion event callbacks (and, similarly, all four touch event callbacks) in production code.

The following snippet shows a pair of motion callback examples. If you test this on a device, you can notice several things. First, the began and ended events happen almost simultaneously from a user perspective. Playing sounds for both types is overkill. Second, there is a bias toward side-to-side shake detection. The iPhone is better at detecting side-to-side shakes than the front-to-back and up-down versions. Finally, Apple's motion implementation uses a slight lockout approach. You cannot generate a new motion event until a second or so after the previous one was processed. This is the same lockout used by Shake to Shuffle and Shake to Undo events:

```
- (void)motionBegan:(UIEventSubtype)motion
    withEvent:(UIEvent *)event {

    // Play a sound whenever a shake motion starts
    if (motion != UIEventSubtypeMotionShake) return;
    [self playSound:startSound];
}

- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
    // Play a sound whenever a shake motion ends
     if (motion != UIEventSubtypeMotionShake) return;
    [self playSound:endSound];
}
```

Recipe: Using External Screens

There are many ways to use external screens. Take the newest iPads, for example. The second and third generation models offer built-in screen mirroring. Attach a VGA or HDMI cable and your content can be shown on external displays and on the built-in screen. Certain devices enable you to mirror screens wirelessly to Apple TV using AirPlay, Apple's proprietary cable-free over-the-air video solution. These mirroring features are extremely handy, but you're not limited to simply copying content from one screen to another in iOS.

The `UIScreen` class enables you to detect and write to external screens independently. You can treat any connected display as a new window and create content for that display separate from any view you show on the primary device screen. You can do this for any wired screen, and starting with the iPad 2 (and later) and the iPhone 4S (and later), you can do so wirelessly using AirPlay to Apple TV 2 (and later). A third-party app called Reflector enables you to mirror your display to Mac or Windows computers using AirPlay.

Geometry is important. Here's why. iOS devices currently include the 320×480 old-style iPhone displays, the 640×960-pixel Retina display units, and the 1024×768-pixel iPads. Typical composite/component output is produced at 720×480 pixels (480i and 480p), VGA at 1024×768 and 1280×720 (720p), and then there's the higher quality HDMI output available as well.

Add to this the issues of overscan and other target display limitations, and Video Out quickly becomes a geometric challenge. Fortunately, Apple has responded to this challenge with some handy real-world adaptations. Instead of trying to create one-to-one correspondences with the output screen and your built-in device screen, you can build content based on the available properties of your output display. You just create a window, populate it, and display it.

If you intend to develop Video Out applications, don't assume that your users are strictly using AirPlay. Many users still connect to monitors and projectors using old-style cable connections. Make sure you have at least one of each type of cable on-hand (composite, component, VGA, and HDMI) and an AirPlay-ready iPhone and iPad, so you can thoroughly test on each output configuration. Third-party cables (typically imported from the Far East, not branded with Made for iPhone/iPad) won't work, so make sure you purchase Apple-branded items.

Detecting Screens

The `UIScreen` class reports how many screens are connected. You know that an external screen is connected whenever this count goes above 1. The first item in the screens array is always your primary device screen:

```
#define SCREEN_CONNECTED ([UIScreen screens].count > 1)
```

Each screen can report its bounds (that is, its physical dimensions in points) and its screen scale (relating the points to pixels). Two standard notifications enable you to observe when screens have been connected to and disconnected from the device.

```
// Register for connect/disconnect notifications
[[NSNotificationCenter defaultCenter]
    addObserver:self selector:@selector(screenDidConnect:)
    name:UIScreenDidConnectNotification object:nil];
[[NSNotificationCenter defaultCenter]
    addObserver:self selector:@selector(screenDidDisconnect:)
    name:UIScreenDidDisconnectNotification object:nil];
```

Connection means *any* kind of connection, whether by cable or via AirPlay. Whenever you receive an update of this type, make sure you count your screens and adjust your user interface to match the new conditions.

It's your responsibility to set up windows whenever new screens are attached and tear them down upon detach events. Each screen should have its own window to manage content for that output display. Don't hold onto windows upon detaching screens. Let them release and then re-create them when new screens appear.

> **Note**
>
> Mirrored screens are not represented in the `screens` array. Instead the mirror is stored in the main screen's `mirroredScreen` property. This property is nil when mirroring is disabled, unconnected, or simply not supported by the device's abilities.
>
> Creating a new screen and using it for independent external display always overrides mirroring. So even if the user has enabled mirroring, when your application begins writing to and creating an external display, it takes priority.

Retrieving Screen Resolutions

Each screen provides an `availableModes` property. This is an array of resolution objects ordered from least-to-highest resolution. Each mode has a `size` property indicating a target pixel-size resolution. Many screens support multiple modes. For example, a VGA display might have as many as one-half dozen or more different resolutions it offers. The number of supported resolutions varies by hardware. There will always be at least one resolution available, but you should offer choices to users when there are more.

Setting Up Video Out

After retrieving an external screen object from the `[UIScreens screens]` array, query the available modes and select a size to use. As a rule, you can get away with selecting the last mode in the list to always use the highest possible resolution, or the first mode for the lowest resolution.

To start a Video Out stream, create a new `UIWindow` and size it to the selected mode. Add a new view to that window for drawing on. Then assign the window to the external screen and make it key and visible. This orders the window to display and prepares it for use. After you do

that, make the original window key again. This allows the user to continue interacting with the primary screen. Don't skip this step. Nothing makes end users more cranky than discovering their expensive device no longer responds to their touches:

```
self.outputWindow = [[UIWindow alloc] initWithFrame:theFrame];
outputWindow.screen = secondaryScreen;
[outputWindow makeKeyAndVisible];
[delegate.view.window makeKeyAndVisible];
```

Adding a Display Link

Display links are a kind of timer that synchronizes drawing to a display's refresh rate. You can adjust this frame refresh time by changing the display link's `frameInterval` property. It defaults to 1. A higher number slows down the refresh rate. Setting it to 2 halves your frame rate. Create the display link when a screen connects to your device. The `UIScreen` class implements a method that returns a display link object for its screen. You specify the target for the display link and a selector to call.

The display link fires on a regular basis, letting you know when to update the Video Out screen. You can adjust the interval up for less of a CPU load, but you get a lower frame rate in return. This is an important trade-off, especially for direct manipulation interfaces that require a high level of CPU response on the device side.

The code you see in Recipe 1-8 uses common modes for the run loop, providing the least latency. You `invalidate` your display link when you are done with it, removing it from the run loop.

Overscanning Compensation

The `UIScreen` class enables you to compensate for pixel loss at the edge of display screens by assigning a value to the `overscanCompensation` property. The techniques you can assign are described in Apple's documentation but basically correspond to whether you want to clip content or pad it with black space.

VIDEOkit

Recipe 1-8 introduces VIDEOkit, a basic external screen client. It demonstrates all the features needed to get up and going with wired and wireless external screens. You establish screen monitoring by calling `startupWithDelegate:`. Pass it the primary view controller whose job it will be to create external content.

The internal `init` method starts listening for screen attach and detach events and builds and tears down windows as needed. An informal delegate method (`updateExternalView:`) is called each time the display link fires. It passes a view that lives on the external window that the delegate can draw onto as needed.

In the sample code that accompanies this recipe, the view controller delegate stores a local color value and uses it to color the external display:

```
- (void) updateExternalView: (UIImageView *) aView
{
    aView.backgroundColor = color;
}

- (void) action: (id) sender
{
    color = [UIColor randomColor];
}
```

Each time the action button is pressed, the view controller generates a new color. When VIDEOkit queries the controller to update the external view, it sets this as the background color. You can see the external screen instantly update to a new random color.

> **Note**
>
> Reflector App ($15/single license, $50/5-computer license, reflectorapp.com) provides an excellent debugging companion for AirPlay, offering a no-wires/no-Apple TV solution that works on Mac and Windows computers. It mimics an Apple TV AirPlay receiver, letting you broadcast from iOS direct to your desktop and record that output.

Recipe 1-8 **VIDEOkit**

```
@interface VIDEOkit : NSObject
{
    UIImageView *baseView;
}
@property (nonatomic, weak)   UIViewController *delegate;
@property (nonatomic, strong) UIWindow *outputWindow;
@property (nonatomic, strong) CADisplayLink *displayLink;
+ (void) startupWithDelegate: (id) aDelegate;
@end

@implementation VIDEOkit
static VIDEOkit *sharedInstance = nil;

- (void) setupExternalScreen
{
    // Check for missing screen
    if (!SCREEN_CONNECTED) return;

    // Set up external screen
    UIScreen *secondaryScreen = [UIScreen screens][1];
    UIScreenMode *screenMode =
```

```
        [[secondaryScreen availableModes] lastObject];
    CGRect rect = (CGRect){.size = screenMode.size};
    NSLog(@"Extscreen size: %@", NSStringFromCGSize(rect.size));

    // Create new outputWindow
    self.outputWindow = [[UIWindow alloc] initWithFrame:CGRectZero];
    _outputWindow.screen = secondaryScreen;
    _outputWindow.screen.currentMode = screenMode;
    [_outputWindow makeKeyAndVisible];
    _outputWindow.frame = rect;

    // Add base video view to outputWindow
    baseView = [[UIImageView alloc] initWithFrame:rect];
    baseView.backgroundColor = [UIColor darkGrayColor];
    [_outputWindow addSubview:baseView];

    // Restore primacy of main window
    [_delegate.view.window makeKeyAndVisible];
}

- (void) updateScreen
{
    // Abort if the screen has been disconnected
    if (!SCREEN_CONNECTED && _outputWindow)
        self.outputWindow = nil;

    // (Re)initialize if there's no output window
    if (SCREEN_CONNECTED && !_outputWindow)
        [self setupExternalScreen];

    // Abort if encountered some weird error
    if (!self.outputWindow) return;

    // Go ahead and update
    SAFE_PERFORM_WITH_ARG(_delegate,
        @selector(updateExternalView:), baseView);
}

- (void) screenDidConnect: (NSNotification *) notification
{
    NSLog(@"Screen connected");
    UIScreen *screen = [[UIScreen screens] lastObject];

    if (_displayLink)
    {
        [_displayLink removeFromRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSRunLoopCommonModes];
```

```
        [_displayLink invalidate];
        _displayLink = nil;
    }

    self.displayLink = [screen displayLinkWithTarget:self
        selector:@selector(updateScreen)];
    [_displayLink addToRunLoop:[NSRunLoop currentRunLoop]
        forMode:NSRunLoopCommonModes];
}

- (void) screenDidDisconnect: (NSNotification *) notification
{
    NSLog(@"Screen disconnected.");
    if (_displayLink)
    {
        [_displayLink removeFromRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSRunLoopCommonModes];
        [_displayLink invalidate];
        self.displayLink = nil;
    }
}

- (id) init
{
    if (!(self = [super init])) return self;

    // Handle output window creation
    if (SCREEN_CONNECTED)
        [self screenDidConnect:nil];

    // Register for connect/disconnect notifications
    [[NSNotificationCenter defaultCenter]
        addObserver:self selector:@selector(screenDidConnect:)
        name:UIScreenDidConnectNotification object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(screenDidDisconnect:)
        name:UIScreenDidDisconnectNotification object:nil];

    return self;
}

- (void) dealloc
{
    [self screenDidDisconnect:nil];
    self.outputWindow = nil;
}
```

```
+ (VIDEOkit *) sharedInstance
{
    if (!sharedInstance)
        sharedInstance = [[self alloc] init];
    return sharedInstance;
}

+ (void) startupWithDelegate: (id) aDelegate
{
    [[self sharedInstance] setDelegate:aDelegate];
}
@end
```

> **Get This Recipe's Code**
>
> To find this recipe's full sample project, point your browser to https://github.com/erica/iOS-6-Advanced-Cookbook and go to the folder for Chapter 1.

Tracking Users

Tracking is an unfortunate reality of developer life. Apple deprecated the `UIDevice` property that provided a unique identifier tied to device hardware. It replaced it with two identifier properties. Use `identifierForAdvertising` to return a device-specific string unique to the current device. The `identifierForVendor` property supplies a string that's tied to each app vendor. This should return the same unique string regardless of which of your apps is in use. This is *not* a customer id. The same app on a different device can return a different string, as can an app from a different vendor.

These identifiers are built using the new `NSUUID` class. You can use this class outside of the tracking scenario to create UUID strings that are guaranteed to be globally unique. Apple writes, "UUIDs (Universally Unique Identifiers), also known as GUIDs (Globally Unique Identifiers) or IIDs (Interface Identifiers), are 128-bit values. A UUID is made unique over both space and time by combining a value unique to the computer on which it was generated and a value representing the number of 100-nanosecond intervals since October 15, 1582 at 00:00:00."

The `UUID` class method can generate a new RFC 4122v4 UUID on demand. Use `[NSUUID UUID]` to return a new instance. (Bonus: It's all in uppercase!) From there, you can retrieve the `UUIDString` representation or request the bytes directly via `getUUIDBytes:`.

One More Thing: Checking for Available Disk Space

The `NSFileManager` class enables you to determine how much space is free on the iPhone and how much space is provided on the device as a whole. Listing 1-1 demonstrates how to check

for these values and show the results using a friendly comma-formatted string. The values returned represent the free space in bytes.

Listing 1-1 **Recovering File System Size and File System Free Size**

```
- (NSString *) commaFormattedStringWithLongLong: (long long) num
{
    // Produce a properly formatted number string
    // Alternatively use NSNumberFormatter
    if (num < 1000)
        return [NSString stringWithFormat:@"%d", num];
    return    [[self commasForNumber:num/1000]
        stringByAppendingFormat:@",%03d", (num % 1000)];
}

- (void) action: (UIBarButtonItem *) bbi
{
    NSFileManager *fm = [NSFileManager defaultManager];
    NSDictionary *fattributes =
        [fm fileSystemAttributesAtPath:NSHomeDirectory()];
    NSLog(@"System space: %@",
        [self commaFormattedStringWithLongLong:[[fattributes
        objectForKey:NSFileSystemSize] longLongValue]]);
    NSLog(@"System free space: %@",
        [self commasForNumber:[[fattributes
        objectForKey:NSFileSystemFreeSize] longLongValue]]);
}
```

Summary

This chapter introduced core ways to interact with an iPhone device. You saw how to recover device info, check the battery state, and subscribe to proximity events. You learned how to differentiate the iPod touch from the iPhone and iPad and determine which model you're working with. You discovered the accelerometer and saw it in use through several examples, from the simple "finding up" to the more complex shake detection algorithm. You jumped into Core Motion and learned how to create update blocks to respond to device events in real time. Finally, you saw how to add external screen support to your applications. Here are a few parting thoughts about the recipes you just encountered:

- The iPhone's accelerometer provides a novel way to complement its touch-based interface. Use acceleration data to expand user interactions beyond the "touch here" basics and to introduce tilt-aware feedback.

- Low-level calls can be App Store-friendly. They don't depend on Apple APIs that may change based on the current firmware release. UNIX system calls may seem daunting, but many are fully supported by the iOS device family.
- Remember device limitations. You may want to check for free disk space before performing file-intensive work and for battery charge before running the CPU at full steam.
- Dive into Core Motion. The real-time device feedback it provides is the foundation for integrating iOS devices into real-world experiences.
- Now that AirPlay has cut the cord for external display tethering, you can use Video Out for many more exciting projects than you might have previously imagined. AirPlay and external video screens mean you can transform your iOS device into a remote control for games and utilities that display on big screens and are controlled on small ones.
- When submitting to iTunes, use your Info.plist file to determine which device capabilities are required. iTunes uses this list of required capabilities to determine whether an application can be downloaded to a given device and run properly on that device.

This page intentionally left blank

Index

A

B

C

D

E

F

G

H

I

J-L

M

N

O

P

Q

R

S

T

U

V

W-Z

This page intentionally left blank