Lee S. Barney

Developing Hybrid Applications for the iPhone

Using HTML, CSS, and JavaScript to Build Dynamic Apps for the iPhone

Developer's Library

Library of Congress Cataloging-in-Publication Data

Barney, Lee.
Developing hybrid applications for the iPhone : using HTML, CSS, and JavaScript to build dynamic apps for the iPhone / Lee S. Barney.

p. cm.

Includes index.

ISBN 978-0-321-60416-3 (pbk. : alk. paper) 1. iPhone (Smartphone)–Programming. 2. Application software–Development. 3. Cross-platform software development. I. Title.

TK6570.M6B37 2009

621.3845'6–dc22

2009019162

ISBN-13: 978-0-321-60416-3
ISBN-10: 0-321-60416-4

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

First printing June 2009

Editor-in-Chief
Karen Gettman

Acquisitions Editor
Chuck Toporek

Development Editor
Sheri Cain

Managing Editor
Kristy Hart

Project Editor
Jovana San Nicolas-Shirley

Copy Editor
Deadline Driven Publishing

Indexer
Erika Millen

Proofreader
Kathy Ruiz

Technical Reviewers
August Trometer
Randall Tamura

Publishing Coordinator
Romny French

Cover Designer
Gary Adair

Compositor
Jake McFarland

Preface

This book shows you how to create a new type of iPhone application: hybrid applications written in HTML, CSS, and JavaScript. Hybrid iPhone applications are standalone applications that run like regular applications on your iPhone, but don't require the files to live on a server on the Internet.

Creating hybrid iPhone applications reduces creation time and the learning curve required to get your application into the hands of your customers, because you don't have to learn Objective-C or have an intimate knowledge of the Cocoa frameworks.

Hybrid Application Development Tools

This book covers the two most commonly used open-source JavaScript software packages for writing applications for the iPhone and iPod touch devices: QuickConnectiPhone and PhoneGap. These packages enable you to build applications that access native device features directly from JavaScript, such as vibration, GPS location information, the accelerometer, and many other things—all without writing a single line of Objective-C or Cocoa.

QuickConnectiPhone, downloaded from http://sourceforge.net/projects/quickconnect, exposes the most native device behavior and provides a highly engineered, full-featured framework for development use. QuickConnectiPhone dramatically reduces your application's time-to-market because part of the framework consists of all of the glue code you have to typically write in Objective-C, Cocoa, and JavaScript. Best of all, it does not require a remote server for hosting JavaScript, HTML, and CSS files.

The second package is PhoneGap, downloaded from http://phonegap.com. PhoneGap exposes fewer native behaviors and is a library rather than a full-fledged framework. As a library, PhoneGap enables you to engineer your application any way you want. It does, however, require a remote server for hosting files.

To reduce the learning curve and improve your understanding, good, solid examples are used throughout this book.

If you want to create installable iPhone applications, have the web skills required, and want to create dynamic, compelling solutions that people will use, this book shows you how using these two packages.

Table P.1 compares what each package can do at the time of writing this book.

Table P.1 **Comparing the Features of QuickConnectiPhone and PhoneGap**

Behavior/Data Available	QuickConnectiPhone	PhoneGap
GPS	Yes	Yes
Accelerometer	Yes	Yes
Vibrate	Yes	Yes
System sounds	Yes	Yes
Ad-hoc (Bonjour) networking	Yes	No
Sync cable networking	Yes	No
Browser-based database access	Yes	No
Shipped database access	Yes	No
Drag-and-drop library	Yes	No
AJAX wrapper	Yes	No
Record/Play audio files	Yes	No
Embedded Google maps	Yes	No
Charts and graphs library	Yes	No

How to Use This Book

Each chapter is organized into two parts. The first part shows you how to use the relevant feature of either QuickConnectiPhone or PhoneGap to accomplish a particular task, such as getting the current geolocation of the device. The second part of the chapter shows how the code behind the JavaScript call is written and how it works. You can decide how deep into the JavaScript and Objective-C you want to delve.

The book is organized as follows:

- Chapter 1, "Developing with Dashcode and Xcode," teaches you how to use Dashcode and Xcode together with QuickConnectiPhone and PhoneGap to quickly create fun-to-use applications that run on the iPhone. This chapter includes basic Dashcode use and methods for moving your Dashcode application into Xcode for compiling and running on devices.
- Chapter 2, "JavaScript Modularity and iPhone Applications," teaches you how to dramatically reduce your time to market by taking advantage of the modularity of the QuickConnectiPhone framework. How front controllers, application controllers, and JavaScript reflection are used in code is explained.
- Chapter 3, "Creating iPhone User Interfaces," helps ensure that Apple App Store distribution approves your applications. It describes best practices for creating highly usable iPhone applications. The different types of applications usually created for iPhones are described as well as pitfalls to watch out for.
- Chapter 4, "GPS, Acceleration, and Other Native Functions with QuickConnectiPhone," shows you how to get GPS, acceleration, and device

description information, and it teaches you how to vibrate your phone and play and record audio files. You use the QuickConnectiPhone framework to access and use these device behaviors. These abilities give your applications a truly native, fun feel.

- Chapter 5, "GPS, Acceleration, and Other Native Functions with PhoneGap," shows you how to get GPS, acceleration, and device description information as well as how to vibrate your phone and play and record audio files. You use the PhoneGap library to access and use these native device behaviors. These abilities give your applications a truly native, fun feel.
- Chapter 6, "Embedding Google Maps," shows you how to put a Google map inside your application using QuickConnectiPhone. This is one of the most requested pieces of functionality and means you won't have to send your users to the map application!
- Chapter 7, "Database Access," shows you how to get information from and store data in SQLite databases included in your application created with the QuickConnectiPhone framework. Do you need to ship a predefined set of data in a database with your new applications? Read this chapter.
- Chapter 8, "Remote Data Access," shows you how to make accessing and using data from remote servers and/or service in your installed application easy with a wrapper that lets you pull information from anywhere. Maybe you need to get data from an online blog and merge it with a Twitter feed. QuickConnectiPhone's remote-data-access module makes it easy.

The following appendices are also included:

- Appendix A, "Introduction to JSON," provides you with a brief introduction to JavaScript Object Notation (JSON). JSON is one of the most commonly used and easiest ways to transfer your data wherever it needs to go.
- Appendix B, "The QuickConnectFamily Development Roadmap," provides an overview of the growth of QuickConnectiPhone in the future. If you plan to create applications for iPhones and other platforms, such as Google's Android phones, Nokia phones, Blackberries, and desktops such as Mac OS X, Linux, and Windows, you should take a look at this appendix.
- Appendix C, "The PhoneGap Development Roadmap," provides an overview of the growth of PhoneGap in the future. If you plan to create applications for iPhones and other platforms, such as Google's Android phones, Nokia phones, Blackberries, and desktops such as Mac OS X, Linux, and Windows, you should take a look at this appendix.

Online Resources

QuickConnectiPhone and PhoneGap are undergoing rapid development. To keep up with the new functions and capabilities and to learn more, use the following links.

QuickConnectiPhone

- Download examples and the framework from https://sourceforge.net/projects/quickconnect/
- Review the development blog at http://tetontech.wordpress.com
- Read the Wiki at http://quickconnect.pbwiki.com/FrontPage
- Find the Google group at http://groups.google.com/group/quickconnectiPhone/
- Twitter at http://twitter.com/quickconnect

PhoneGap

- Download examples and the framework from https://sourceforge.net/projects/phonegapinstall/
- Visit the web site at http://www.phonegap.com/
- Read the Wiki at http://phonegap.pbwiki.com/
- Find the Google group at http://groups.google.com/group/phonegap
- Twitter at http://twitter.com/phonegap

Prerequisites

You need a basic understanding of HTML, CSS, and JavaScript to effectively use this book. If you have created web pages using these tools, you are well on your way to creating iPhone applications. If you need help with Objecive-C in both QuickConnectiPhone and PhoneGap, it is provided. This book is not intended to be an introductory book on Objective-C or how to use it to develop iPhone applications.

You need to download and install Apple's Xcode tools from the iPhone developer web site at http://developer.apple.com/iphone. This requires Mac OS X 10.5 or greater and an Intel-based Mac.

Although it isn't required, you should also have either an iPhone or an iPod touch, so you can test and run the applications on those devices.

Acknowledgments

A special thanks to Daniel Barney for working through and debugging the embedded Google maps code. Thanks also to my coworkers in the Brigham Young University–Idaho Computer Information Technology Department for listening and giving suggestions.

About the Author

Lee S. Barney (Rexburg, Idaho) is a professor at Brigham Young University–Idaho in the Computer Information Technology Department of the Business and Communication College. He has worked as CIO and CTO of @HomeSoftware, a company that produced web-based, mobile data, and scheduling applications for the home health care industry. Prior to this, he worked for more than seven years as a programmer, senior software engineer, quality assurance manager, development manager, and project manager for AutoSimulations, Inc., the leading supplier of planning and scheduling software to the semiconductor industry. He is the author of *Oracle Database AJAX & PHP Web Application Development*.

Contacting the Author

To contact the author by email, use quickconnectfamily@gmail.com. For other types of contact, use Twitter, the Wiki, and Google Group links provided earlier.

4

GPS, Acceleration, and Other Native Functions with QuickConnect

The iPhone has many unique capabilities that you can use in your applications. These capabilities include vibrating the phone, playing system sounds, accessing the accelerometer, and using GPS location information. It is also possible to write debug messages to the Xcode console when you write your application. Accessing these capabilities is not limited to Objective-C applications. Your hybrid applications can do these things from within JavaScript. The first section of this chapter explains how to use these and other native iPhone functionalities with the QuickConnect JavaScript API. The second section shows the Objective-C code underlying the QuickConnect JavaScript Library.

Section 1: JavaScript Device Activation

The iPhone is a game-changing device. One reason for this is that access to hardware such as the accelerometer that is available to people creating applications. These native iPhone functions enable you to create innovative applications. You decide how your application should react to a change in the acceleration or GPS location. You decide when the phone vibrates or plays some sort of audio.

The QuickConnectiPhone com.js file has a function that enables you to access this behavior in a simple, easy-to-use manner. The `makeCall` function is used in your application to make requests of the phone. To use `makeCall`, you need to pass two parameters. The first is a command string and the second is a string version of any parameters that might be needed to execute the command. Table 4.1 lists each standard command, the parameters required for it, and the behavior of the phone when it acts on the command.

Table 4.1 **MakeCall Commands API**

Command String	Message String	Behavior
logMessage	Any information to be logged in the Xcode terminal.	The message appears in the Xcode terminal when the code runs.
rec	A JSON string of a JavaScript array containing the name of the audio file to create as the first element. The second element of the array is either `start` or `stop` depending on if your desire is to start or stop recording audio data.	A caf audio file with the name defined in the message string is created.
play	A JSON string of a JavaScript array containing the name of the audio file to be played as the first element. The second element of the array is either `start` or `stop` depending on if your desire is to start or stop playing the audio file.	The caf audio file, if it exists, is played through the speakers of the device or the headphones.
loc	None	The Core Location behavior of the device is triggered and the latitude, longitude, and altitude information are passed back to your JavaScript application.
playSound	–1	The device vibrates.
playSound	0	The laser audio file is played.
showDate	DateTime	The native date and time picker is displayed.
showDate	Date	The native date picker is displayed.

The DeviceCatalog sample application includes a `Vibrate` button, which when clicked, causes the phone to shake. The button's `onclick` event handler function is called `vibrateDevice` and is seen in the following example. This function calls the `makeCall` function and passes the `playSound` command with –1 passed as the additional parameter. This call causes the phone to vibrate. It uses the `playSound` command because the iPhone treats vibrations and short system sounds as sounds.

```
function vibrateDevice(event)
{
    //the -1 indicator causes the phone to vibrate
    makeCall("playSound", -1);
}
```

Because vibration and system sounds are treated the same playing a system sound is almost identical to vibrating the phone. The Sound button's `onclick` event handler is called `playSound`. As you can see in the following code, the only difference between it and `vibrateDevice` is the second parameter.

If a 0 is passed as the second parameter, the laser.wav file included in the DeviceCatalog project's resources is played as a system sound. System sound audio files must be less than five seconds long or they cannot be played as sounds. Audio files longer than this are played using the `play` command, which is covered later in this section.

```
function playSound(event)
{
    //the 0 indicator causes the phone to play the laser sound
    makeCall("playSound", 0);
}
```

The `makeCall` function used in the previous code exists completely in JavaScript and can be seen in the following code. The `makeCall` function consists of two portions. The first queues up the message if it cannot be sent immediately. The second sends the message to underlying Objective-C code for handling. The method used to pass the message is to change the `window.location` property to a nonexistent URL, `call`, with both parameters passed to the function as parameters of the URL.

```
function makeCall(command, dataString){
    var messageString = "cmd="+command+"&msg="+dataString;
    if(storeMessage || !canSend){
        messages.push(messageString);
    }
    else{
        storeMessage = true;
        window.location = "call?"+messageString;
    }
}
```

Setting the URL in this way causes a message, including the URL and its parameters, to be sent to an Objective-C component that is part of the underlying QuickConnectiPhone framework. This Objective-C component is designed to terminate the loading of the new page and pass the command and the message it was sent to the framework's command-handling code. To see how this is done, see Section 2.

The `playSound` and the `logMessage`, `rec`, and `play` commands are unidirectional, which means that communication from JavaScript to Objective-C with no data expected back occurs. The remaining unidirectional standard commands all cause data to be sent from the Objective-C components back to JavaScript.

The passing of data back to JavaScript is handled in two ways. An example of the first is used to transfer acceleration information in the x, y, and z coordinates by a call to the `handleRequest` JavaScript function, described in Chapter 2, "JavaScript Modularity and iPhone Applications." The call uses the `accel` command and the x, y, and z

coordinates being passed as a JavaScript object from the Objective-C components of the framework.

The mappings.js file indicates that the `accel` command is mapped to the `displayAccelerationVCF` function, as shown in the following line.

```
mapCommandToVCF('accel', displayAccelerationVCF);
```

This causes `displayAccelerationVCF` to be called each time the accelerometers detect motion. This function is responsible for handling all acceleration events. In the DeviceCatalog example application, the function simply inserts the x, y, and z acceleration values into an HTML div. You should change this function to use these values for your application.

The second way to send data back to JavaScript uses a call to the `handleJSONRequest` JavaScript function. It works much like the `handleRequest` function described in Chapter 2, but expects a JSON string as its second parameter. This function is a façade for the `handleRequest` function. As shown in the following code, it simply converts the JSON string that is its second parameter into a JavaScript object and passes the command and the new object to the `handleRequest` method. This method of data transfer is used to reply to a GPS location request initiated by a `makeCall("loc")` call and the request to show a date and time picker.

```
function handleJSONRequest(cmd, parametersString){
    var paramsArray = null;
    if(parametersString){
        var paramsArray = JSON.parse(parametersString);
    }
    handleRequest(cmd, paramsArray);
}
```

In both cases, the resulting data is converted to a JSON string and then passed to `handleJSONRequest`. For more information on JSON, see Appendix A, "Introduction to JSON."

Because JSON libraries are available in both JavaScript and Objective-C, JSON becomes a good way to pass complex information between the two languages in an application. A simple example of this is the onclick handlers for the starting and stopping of recording and playing back audio files.

The `playRecording` handler is typical of all handlers for the user interface buttons that activate device behaviors. As shown in the following example, it creates a JavaScript array, adds two values, converts the array to a JSON string, and then executes the `makeCall` function with the `play` command.

```
function playRecording(event)
{
    var params = new Array();
    params[0] = "recordedFile.caf";
    params[1] = "start";
    makeCall("play", JSON.stringify(params));
}
```

To stop playing a recording, a `makeCall` is also issued with the `play` command, as shown in the previous example, but instead of the second param being `start`, it is set to `stop`. The `terminatePlaying` function in the main.js file implements this behavior.

Starting and stopping the recording of an audio file is done in the same way as `playRecording` and `terminatePlaying` except that instead of the `play` command, `rec` is used. Making the implementation of the starting and stopping of these related capabilities similar makes it much easier for you to add these behaviors to your application.

As seen earlier in this section, some device behaviors, such as vibrate require communication only from the JavaScript to the Objective-C handlers. Others, such as retrieving the current GPS coordinates or the results of a picker, require communication in both directions. Figure 4.1 shows the DeviceCatalog application with GPS information.

Figure 4.1 The DeviceCatalog example application showing GPS information.

As with some of the unidirectional examples already examined, communication starts in the JavaScript of your application. The `getGPSLocation` function in the main.js file initiates the communication using the `makeCall` function. Notice that as in the earlier examples, `makeCall` returns nothing. `makeCall` uses an asynchronous communication

protocol to communicate with the Objective-C side of the library even when the communication is bidirectional, so no return value is available.

```
function getGPSLocation(event)
{
    document.getElementById('locDisplay').innerText = '';
    makeCall("loc");
}
```

Because the communication is asynchronous, as AJAX is, a callback function needs to be created and called to receive the GPS informartion. In the QuickConnectiPhone framework, this is accomplished by creating a mapping in the mapping file that maps the command `showLoc` to a function:

```
mapCommandToVCF('showLoc', displayLocationVCF);
```

In this case, it is mapped to the `displayLocationVCF` view control function. This simple example function is used only to display the current GPS location in a div on the screen. Obviously, these values can also be used to compute distances to be stored in a database or to be sent to a server using the ServerAccessObject described in Chapter 8, "Remote Data Access."

```
function displayLocationVCF(data, paramArray){
    document.getElementById('locDisplay').innerText = 'latitude:
'+paramArray[0]+'\nlongitude: '+paramArray[1]+'\naltitude:
'+paramArray[2];
}
```

Displaying a picker, such as the standard date and time picker, and then displaying the selected results is similar to the previous example. This process also begins with a call from JavaScript to the device-handling code. In this case, the event handler function of the button is the `showDateSelector` function found in the main.js file.

```
function showDateSelector(event)
{
    makeCall("showDate", "DateTime");
}
```

As with the GPS example, a mapping is also needed. This mapping maps the `showPickResults` command to the `displayPickerSelectionVCF` view control function, as shown in the following:

```
mapCommandToVCF('showPickResults', displayPickerSelectionVCF);
```

The function to which the command is mapped inserts the results of the user's selection in a simple div, as shown in the following code. Obviously, this information can be used in many ways.

```
function displayPickerSelectionVCF(data, paramArray){
    document.getElementById('pickerResults').innerHTML = paramArray[0];
```

Some uses of `makeCall`, such as the earlier examples in this section, communicate unidirectionally from the JavaScript to the Objective-C device handlers. Those just examined use bidirectional communication to and from handlers. Another type of communication that is possible with the device is unidirectionally from the device to your JavaScript code. An example of this is accelerometer information use.

The Objective-C handler for acceleration events, see Section 2 to see the code, makes a JavaScript `handleRequest` call directly passing the `accel` command. The following `accel` command is mapped to the `displayAccelerationVCF` view control function.

```
mapCommandToVCF('accel', displayAccelerationVCF);
```

As with the other VCFs, this one inserts the acceleration values into a div.

```
function displayAccelerationVCF(data, param){
    document.getElementById('accelDisplay').innerText ='x:
'+param.x+'\ny: '+param.y+'\nz: '+param.z;
}
```

One difference between this function and the others is that instead of an array being passed, this function has an object passed as its `param` parameter. Section 2 shows how this object was created from information passed from the Objective-C acceleration event handler.

This section has shown you how to add some of the most commonly requested iPhone behaviors to your JavaScript-based application. Section 2 shows the Objective-C portions of the framework that support this capability.

Section 2: Objective-C Device Activation

This section assumes you are familiar with Objective-C and how it is used to create iPhone applications. If you are not familiar with this, Erica Sadun's book *The iPhone Developer's Cookbook* is available from Pearson Publishing. If you just want to use the QuickConnectiPhone framework to write JavaScript applications for the iPhone, you do not have to read this section.

Using Objective-C to vibrate the iPhone is one of the easiest behaviors to implement. It can be done with the following single line of code if you include the AudioToolbox framework in the resources of your project.

```
AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);
```

The question then becomes, "How can I get the `AudioServicesPlaySystemSound` function to be called when the UIWebView is told to change its location?"

The QuickConnectViewController implements the `shouldStartLoadWithRequest` delegate method. Because the delegate of the embedded UIWebView, called `aWebView`, is set to be the QuickConnectViewController this method is called every time the embedded UIWebView is told to change its location. The following code and line 90 of the QuickConnectViewController.m file show this delegate being set.

```
[aWebView setDelegate:self];
```

The basic behavior of the `shouldStartLoadWithRequest` function is straightforward. It is designed to enable you to write code that decides if the new page requested should actually be loaded. The QuickConnectiPhone framework takes advantage of the decision-making capability to disallow page loading by any of the requests made by the JavaScript calls shown in Section 1 and execute other Objecive-C code.

The `shouldStartLoadWithRequest` method has several parameters that are available for use. These include

- `curWebView`—The UIWebView containing your JavaScript application.
- `request`—A NSURLRequest containing the new URL among other items.
- `navigationType`—A UIWebViewNavigationType that can be used to determine if the request is the result of the user selecting a link or if it was generated as a result of some other action.

```
-(BOOL)webView:(UIWebView *)curWebView
   shouldStartLoadWithRequest:(NSURLRequest *)request
navigationType:(UIWebViewNavigationType)navigationType
```

The URL assembled by the `makeCall` JavaScript function that causes the device to vibrate, `call?cmd=playSound&msg=-1` is contained in the `request` object and is easily retrieved as a string by passing the `URL` message to it. This message returns an NSURL-type object, which is then passed the `absoluteString` message. Thus, an NSString pointer representing the URL is obtained. This string, seen as `url` in the following code, can then be split into an array using the ? as the splitting delimiter, yielding an array of NSString pointers.

```
NSString *url = [[request URL] absoluteString];
NSArray *urlArray = [url componentsSeparatedByString:@"?"];
```

`urlArray` contains two elements. The first is the `call` portion of the URL and the second is the command string `cmd=playSound&msg=-1`. To determine which command to act on and any parameters that might need to be used, in this case the –1, the command string requires further parsing. This is done by splitting the `commandString` at the & character. This creates another array called `urlParamsArray`.

```
NSString *commandString = [urlArray objectAtIndex:1];
NSArray *urlParamsArray = [commandString
componentsSeparatedByString:@"&"];
//the command is the first parameter in the URL
cmd = [[[urlParamsArray objectAtIndex:0]
componentsSeparatedByString:@"="] objectAtIndex:1];
```

In this case, requesting that the device to vibrate, the first element of the `urlParamsArray` array becomes `cmd=playSound` and the second is `msg=-1`. Thus, splitting the elements of the urlParamsArray can retrieve the command to be executed and the parameter. The = character is the delimiter to split each element of the `urlParamsArray`.

Lines 1– 3 in the following example retrieve the parameter sent as the value associated with the `msg` key in the URL as the NSString `parameterArrayString`. Because the JavaScript that assembled the URL converts all items that are this value to JSON, this NSString is an object that has been converted into JSON format. This includes numbers, such as the current example, and strings, arrays, or other parameters passed from the JavaScript. Additionally, if spaces or other special characters appear in the data, the UIWebView escapes them as part of the URL. Therefore, lines 6–8 in the following code is needed to unescape any special characters in the JSON string.

```
1 NSString *parameterArrayString = [[[urlParamsArray
2   objectAtIndex:1] componentsSeparatedByString:@"="]
3   objectAtIndex:1];
4   //remove any encoding added as the UIWebView has
5   //escaped the URL characters.
6   parameterArrayString = [parameterArrayString
7     stringByReplacingPercentEscapesUsingEncoding:
8     NSASCIIStringEncoding];
9   SBJSON *generator = [SBJSON alloc];
10   NSError *error;
11   paramsToPass = [[NSMutableArray alloc]
12   initWithArray:[generator
13      objectWithString:parameterArrayString
14      error:&error]];
15   if([paramsToPass count] == 0){
16       //if there was no array of data sent then it must have
17     //been a string that was sent as the only parameter.
18     [paramsToPass addObject:parameterArrayString];
19   }
20   [generator release];
```

Lines 9–14 in the previous code contain the code to convert the JSON string `parameterArrayString` to a native Objective-C NSArray. Line 9 allocates a SBJSON generator object. The generator object is then sent the `objectWithString` message seen in the following:

```
- (id)objectWithString:(NSString*)jsonrep error:(NSError**)error;
```

This multipart message is passed a JSON string, in this case `parameterArrayString`, and an NSError pointer `error`. The `error` pointer is assigned if an error occurs during the conversion process. If no error happens, it is nil.

The return value of this message is in this case the number –1. If a JavaScript array is stringified, it is an NSArray pointer, or if it is a JavaScript string, it is an NSString pointer. If a JavaScript custom object type is passed, the returned object is an NSDictionary pointer.

At this point, having retrieved the command and any parameters needed to act on the command, it is possible to use an `if` or `case` statement to do the actual computation.

Such a set of conditionals is, however, not optimal because they have to be modified each time a command is added or removed. In Chapter 2, this same problem is solved in the JavaScript portion of the QuickConnectiPhone architecture by implementing a front controller function called `handleRequest` that contains calls to implementations of application controllers. Because the problem is the same here, an Objective-C version of `handleRequest` should solve the current problem. Section 3 covers the implementation of the front controllers and application controllers in Objective-C. The following line of code retrieves an instance of the QuickConnect object and passes it the `handleRequest withParameters` multimessage. No further computation is required within the `shouldStartLoadWithRequest` delegate method.

```
[[QuickConnect getInstance] handleRequest:cmd withParameters:paramsToPass];
```

Because the QuickConnect objects' `handleRequest` message is used, there must be a way of mapping the command to the required functionality as shown in Chapter 2 using JavaScript. The `QCCommandMappings` object found in the QCCommandMappings.m and .h files of the QCObjC group contains all the mappings for Business Control Objects (BCO) and View Control Objects (VCO) for this example.

The following code is the `mapCommands` method of the `QCCommandMappings` object that is called when the application starts. It is passed an implementation of an application controller that is used to create the mappings of command to functionality. An explanation of the code for the `mapCommandToVCO` message and the call of `mapCommands` are found in Section 3.

```
1 + (void) mapCommands:(QCAppController*)aController{
2   [aController mapCommandToVCO:@"logMessage" withFunction:@"LoggingVCO"];
3   [aController mapCommandToVCO:@"playSound" withFunction:@"PlaySoundVCO"];
4   [aController mapCommandToBCO:@"loc" withFunction:@"LocationBCO"];
5   [aController mapCommandToVCO:@"sendloc" withFunction:@"LocationVCO"];
6   [aController mapCommandToVCO:@"showDate" withFunction:@"DatePickerVCO"];
7   [aController mapCommandToVCO:@"sendPickResults"
withFunction:@"PickResultsVCO"];
8   [aController mapCommandToVCO:@"play" withFunction:@"PlayAudioVCO"];
9   [aController mapCommandToVCO:@"rec" withFunction:@"RecordAudioVCO"];
10 }
```

Line 3 of the previous code is pertinent to the current example of vibrating the device. As seen earlier in this section, the command received from the JavaScript portion of the application is `playSound`. By sending this command as the first parameter of the `mapCommandToVCO` message and `PlaySoundVCO` as the parameter for the second portion, `withFunction`, a link is made that causes the application controller to send a `doCommand` message with the –1 parameter to the `PlaySoundVCO` class. As you can see, all the other commands in the DeviceCatalog example that are sent from JavaScript are mapped here.

The code for the `PlaySoundVCO` to which the `playSound` command is mapped is found in the PlaySoundVCO.m and PlaySoundVCO.h files. The `doCommand` method contains all the object's behavior.

To play a system sound, a predefined sound, of which vibrate is the only one at the time of writing this book, must be used or a system sound must be generated from a sound file. The `doCommand` of the `PlaySoundVCO` class shows examples of both of these types of behavior.

```
1 + (id) doCommand:(NSArray*) parameters{
2     SystemSoundID aSound =
3       [((NSNumber*)[parameters objectAtIndex:1]) intValue];
4     if(aSound == -1){
5         aSound = kSystemSoundID_Vibrate;
6     }
7     else{
8         NSString *soundFile =
9          [[NSBundle mainBundle] pathForResource:@"laser"
10                ofType:@"wav"];
11         NSURL *url = [NSURL fileURLWithPath:soundFile];
12         //if the audio file is takes to long to play
13         //you will get a -1500 error
14         OSStatus error = AudioServicesCreateSystemSoundID(
15          (CFURLRef) url, &aSound );
16     }
17     AudioServicesPlaySystemSound(aSound);
18     return nil;
19 }
```

As seen in line 4 in the previous example, if the parameter with the index of 1 has a value of –1, the `SystemSoundID aSound` variable is set to the defined `kSystemSoundID_Vibrate` value. If it is not, a system sound is created from the laser.wav file found in the resources group of the application, and the `aSound` variable is set to an identifier generated for the new system sound.

In either case, the C function `AudioServicesPlaySystemSound` is called and the sound is played or the device vibrates. If the device is an iPod Touch, requests for vibration are ignored by the device. In an actual application that has multiple sounds, this function can easily be expanded by passing other numbers as indicators of which sound should be played.

Because the `SystemSoundID` type variable is actually numeric, the system sounds should be generated at application start and the SystemSoundIDs for each of them should be passed to the JavaScript portion of the application for later use. This avoids the computational load of recreating the system sound each time a sound is required, and therefore, increases the quality of the user's experience because there is no delay of the playing of the sound.

Having now seen the process of passing commands from JavaScript to Objective-C and how to vibrate the device or play a short sound, it is now easy to see and understand how to pass a command to Objective-C and have the results returned to the JavaScript portion of the application.

Because these types of communication behave similarly, GPS location detection, which is a popular item in iPhone applications, is shown as an example. It uses this bidirectional, JavaScript-Objective-C communication capability of the QuickConnectiPhone framework.

As with the handling of all the commands sent from the JavaScript framework, there must be a mapping of the `loc` command so that the data can be retrieved and a response sent back.

```
[aController mapCommandToBCO:@"loc" withFunction:@"LocationBCO"];
[aController mapCommandToVCO:@"sendloc" withFunction:@"LocationVCO"];
```

In this case, there are two mappings: The first is to a BCO and the second is to a VCO. As discussed in Chapter 2, BCOs do data retrieval and VCOs are used for data presentation.

Because BCOs for a given command are executed prior to all of the VCOs by the QuickConnectiPhone framework, a `doCommand` message is first sent to the `LocationBCO` class, which retrieves and returns the GPS data. The following `doCommand` method belongs to the `LocationBCO` class. It makes the calls required to get the device to begin finding its GPS location.

```
+ (id) doCommand:(NSArray*) parameters{
    QuickConnectViewController *controller = (QuickConnectViewController*)[parame-
ters objectAtIndex:0];
    [[controller locationManager] startUpdatingLocation];
    return nil;
}
```

This method starts the GPS location hardware by retrieving the first item in the parameter's array that is passed into the method and informing it to start the hardware. The framework always sets the first parameter to be the `QuickConnectViewController` so that it can be used if needed by BCOs or VCOs associated with any command. In all of the Objective-C BCOs and VCOs any parameters sent from JavaScript begin with an index of 1.

The `QuickConnectViewController` object has a built in `CLLocationManager` attribute called `locationManager` that is turned on and off as needed by your application. It is important not to leave this manager running any longer than needed because it uses large amounts of battery power. Therefore, the previous code turns the location hardware on by sending it a `startUpdatingLocation` message each time a location is needed. The location hardware is turned off once the location is found.

`CLLocationManager` objects behave in an asynchronous manner. This means that when a request is made for location information, a predefined callback function is called after the location has been determined. This predefined function allows you access to the location manager and two locations: a previously determined location and a current location.

The location manager works by gradually refining the device's location. As it does this, it calls `didUpdateToLocation` several times. The following code example finds out how long it takes to determine the new location. Line 9 determines if this is less than 5.0 seconds and if it is terminates the location search.

```
1  (void)locationManager:(CLLocationManager *)manager
2      didUpdateToLocation:(CLLocation *)newLocation
3        fromLocation:(CLLocation *)oldLocation
4 {
5     // If it's a relatively recent event, turn off updates to save power
6     NSDate* eventDate = newLocation.timestamp;
7     NSTimeInterval howRecent =
8             [eventDate timeIntervalSinceNow];
9     if (abs(howRecent) < 5.0){
10         [manager stopUpdatingLocation];
11         NSMutableArray *paramsToPass =
12           [[NSMutableArray alloc] initWithCapacity:2];
13         [paramsToPass addObject:self];
14         [paramsToPass addObject:newLocation];
15         [[QuickConnect getInstance]
16             handleRequest:@"sendloc"
17               withParameters:paramsToPass];
18     }
19     // else skip the event and process the next one.
20 }
```

Having terminated the location search, the code then sends a message to the QuickConnect front controller class stating that it should handle a `sendloc` request with the `QuickConnectViewController`, `self`, and the new location passed as an additional parameter.

The `sendloc` command is mapped to the `LocationVCO` handler whose `doCommand` method is seen in the following example. This method retrieves the UIWebView called `webView` from the `QuickConnectViewController` that made the original request for GPS location information. It then places the GPS information into the NSArray called `passingArray`.

To pass the GPS information back to the `webView` object, the NSArray within which it is contained must be converted into a JSON string. The same SBJSON class used earlier to create an array from a JSON string is now used to create a NSString from the NSArray. This is done on lines 21 and 22:

```
1 + (id) doCommand:(NSArray*) parameters{
2     QuickConnectViewController *controller =
3    (QuickConnectViewController*)[parameters
4      objectAtIndex:0];
5     UIWebView *webView = [controller webView];
6     CLLocation *location = (CLLocation*)[parameters
7       objectAtIndex:1];
```

```
8
9     NSMutableArray *passingArray = [[NSMutableArray alloc]
10      initWithCapacity:3];
11    [passingArray addObject: [NSNumber numberWithDouble:
12    location.coordinate.latitude]];
13    [passingArray addObject: [NSNumber numberWithDouble:
14     location.coordinate.longitude]];
15    [passingArray addObject: [NSNumber numberWithFloat:
16    location.altitude]];
17
18    SBJSON *generator = [SBJSON alloc];
19
20    NSError *error;
21    NSString *paramsToPass = [generator
22     stringWithObject:passingArray error:&error];
23    [generator release];
24    NSString *jsString = [[NSString alloc]
25    initWithFormat:@"handleJSONRequest('showLoc', '%@')",
26      paramsToPass];
27  [webView
28    stringByEvaluatingJavaScriptFromString:jsString];
29    return nil;
30 }
```

After converting the GPS location information into a JSON string representing an array of numbers, a call is made to the JavaScript engine inside the `webView` object. This is done by first creating an NSString that is the JavaScript to be executed. In this example, it is a `handleJSONRequest` that is passed `showLoc` as the command and the JSON GPS information as a string. As seen in Section 1, this request causes the GPS data to appear in a div in the HTML page being displayed.

Having seen this example, you can now look at the `DatePickerVCO` and `PickResultsVCO` in the DeviceCatalog example and see how this same approach is used to display the standard date and time selectors, called pickers, that are available in Objective-C. Although predefined pickers available using JavaScript within the UIWebView, they are not as nice from the user's point of view as the standard ones available from within Objective-C. By using these standard ones and any custom ones you may choose to define, your hybrid application will have a smoother user experience.

Section 3: Objective-C Implementation of the QuickConnectiPhone Architecture

The code shown in Sections 1 and 2 depends heavily on an implementation in Objective-C of the same architecture, which is explained in Chapter 2. This section shows how to implement the architecture in Objective-C. To see a full explanation of each component, see Chapter 2, which contains the JavaScript implementation.

As in the JavaScript implementation, all requests for application behavior are handled via a front controller. The front controller is implemented as the class `QuickConnect`, the source for which is found in the QuickConnect.m and QuickConnect.h files. Because messages sent to `QuickConnect` might need to be made from many different locations throughout an application, this class is a singleton.

Singleton classes are written so that only one instantiated object of that class can be allocated within an application. If done correctly, there is always a way to obtain a pointer to this single object from anywhere in the application. With the `QuickConnect` singleton object, this is accomplished by implementing a class method `getInstance` that returns the single `QuickConnect` instance that is allocated the first time this method is called.

Because it is a class method, a `getInstance` message can be sent to the class without instantiating a `QuickConnect` object. When called, it returns a pointer to the underlying `QuickConnect` instance. As seen in the following code, this is accomplished by assigning an instance of the class to a statically defined `QuickConnect` pointer.

```
+ (QuickConnect*)getInstance{
    //since this line is declared static
    //it will only be executed once.
    static QuickConnect *mySelfQC = nil;

    @synchronized([QuickConnect class]) {
        if (mySelfQC == nil) {
            mySelfQC = [QuickConnect singleton];
            [mySelfQC init];
        }
    }
    return mySelfQC;
}
```

The singleton message sent prior to init uses the behavior defined in the `QuickConnect` objects' superclass `FTSWAbstractSingleton`. This superclass allocates the embedded singleton behavior such as overriding new, clone, and other methods that someone might incorrectly attempt to use to allocate another `QuickConnect` instance. Because of this, only the `getInstance` method can be used to create and use a `QuickConnect` object. As with all well-formed objects in Objective-C, after a `QuickConnect` object has been allocated, it must be initialized.

Both the allocation and initialization of the object happen only if no `QuickConnect` object has been assigned to the `mySelfQC` attribute. Additionally, because of the synchronization call surrounding the check for the instantiated `QuickConnect` object, the checking and initialization are thread safe.

`- (void) handleRequest: (NSString*) aCmd withParameters:(NSArray*) parameters` is another method of the `QuickConnect` class. Just as with the JavaScript `handleRequest(aCmd, parameters)` function from Chapter 2, this method is the way to request functionality be executed in your application.

A command string and an array of parameters are passed to the method. In the following example, lines 3–9 show that a series of messages are sent to the application controller. Lines 3 and 4 first execute any VCOs associated with the command. If the command and parameters pass validation, any BCOs associated with the command are executed by a `dispatchToBCO` message. This message returns an `NSMutableArray` that contains the original `parameters` array data to which has been added any data accumulated by any BCO object that might have been called.

```
1 - (void) handleRequest: (NSString*) aCmd
2          withParameters:(NSArray*) parameters{
3   if([self->theAppController dispatchToValCO:aCmd
4      withParameters:parameters] != nil){
5     NSMutableArray *newParameters =
6       [self->theAppController dispatchToBCO:aCmd
7              withParameters:parameters];
8        [self->theAppController dispatchToVCO:aCmd
9              withParameters:newParameters];
10   }
11 }
```

After the completion of the call to `dispatchToBCO:withParameters`, a `dispatchToVCO:withParameters` message is sent. This causes any VCOs also associated with the given command to be executed.

By using the `handleRequest:withParameters` method for all requests for functionality, each request goes through a three-step process.

1. Validation.
2. Execution of business rules (BCO).
3. Execution of view changes (VCO).

As in the JavaScript implementation, each `dispatchTo` method is a façade. In this case, the underlying Objective-C method is `dispatchToCO:withParameters`.

This method first retrieves all the command objects associated with the `default` command in `aMap` the passed parameter. `aMap` contains either BCOs, VCOs, or ValCOs depending on which façade method was called. These default command objects, if any, are retrieved and used for all commands. If you want to have certain command objects used for all commands, you do not need to map them to each individual command. Map them to the `default` command once instead.

For the retrieved command objects to be used, they must be sent a message. The message to be sent is `doCommand`. Lines 19–23 in the following example show this message being retrieved as a selector and the `performSelector` message being passed. This causes the `doCommand` message you have implemented in your `QCCommandObject` to be executed.

```
1 - (id) dispatchToCO: (NSString*)command withParameters:
2         (NSArray*)parameters andMap:(NSDictionary*)aMap{
3      //create a mutable array that contains all of
```

```
4      // the existing parameters.
5      NSMutableArray *resultArray;
6      if(parameters == nil){
7         resultArray = [[NSMutableArray alloc]
8                              initWithCapacity:0];
9     }
10     else{
11        resultArray = [NSMutableArray
12                              arrayWithArray:parameters];
13     }
14     //set the result to be something so
15     //that if no mappings are made the
16     //execution will continue.
17     id result = @"Continue";
18     if([aMap objectForKey:@"default"] != nil){
19        SEL aSelector = @selector(doCommand);
20        while((result = [((QCCommandObject*)
21            [aMap objectForKey:@"default"])
22            performSelector:aSelector
23                withObject:parameters]) != nil){
24            if(aMap == self->businessMap){
25              [resultArray addObject:result];
26            }
27        }
28      }
29      //if all of the default command objects' method calls
30      //return something, execute all of the custom ones.
31      if(result != nil && [aMap objectForKey:command] !=
32           nil){
33         NSArray *theCommandObjects =
34             [aMap objectForKey:command];
35         int numCommandObjects = [theCommandObjects count];
36         for(int i = 0; i < numCommandObjects; i++){
37            QCCommandObject *theCommand =
38               [theCommandObjects objectAtIndex:i];
39            result = [theCommand doCommand:parameters];
40            if(result == nil){
41                resultArray = nil;
42                break;
43            }
44            if(aMap == self->businessMap){
45               [resultArray addObject:result];
46            }
47        }
48    }
49    if(aMap == self->businessMap){
```

```
50        return resultArray;
51    }
52    return result;
53 }
```

After all the `doCommand` messages are sent to any `QCCommandObjects` you mapped to the `default` command, the same is done for `QCCommandObjects` you mapped to the command passed into the method as a parameter. These `QCCommandObjects` have the same reasons for existence as the control functions in the JavaScript implementation. Because `QCCommandObjects` contain all the behavior code for your application, an example is of one is helpful in understanding how they are created.

`QCCommandObject` is the parent class of `LoggingVCO`. As such, `LoggingVCO` must implement the `doCommand` method. The entire contents of the LoggingVCO.m file found in the DeviceCatalog example follows. Its `doCommand` method writes to the log file of the

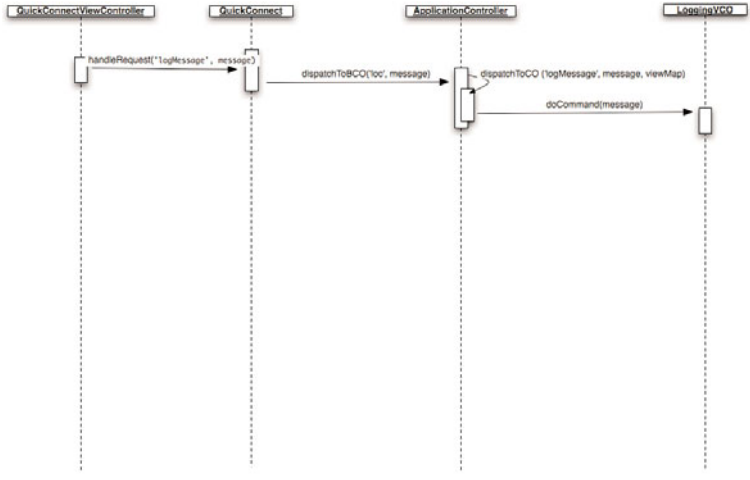

Figure 4.2 A sequence diagram shows the methods called in Objective-C to handle a request to log a JavaScript debug message.

running application. This VCO logs debug messages generated from within the JavaScript code of your application. Figure 4.2 shows the calls required to accomplish this.

The `doCommand` method of the `LoggingVCO` class is small. All `doCommand` methods for the different types of command objects should always be small. They should do one thing only and do it well. If you find that a `doCommand` method you are working on is getting large, you might want to consider splitting it into logical components and creating more than one command object class. The reason for this is that if these methods become long, they are probably doing more than one thing.

In the following example, the "one thing" the `LoggingVCO` does is log messages to the debug console in Xcode. Obviously, this small component can be reused with many commands in combination with other command objects.

The behavior of this VCO consists of a single line that executes the `NSLog` function. In doing so, the first object in the parameters array is appended to a static string and written out.

```
#import "LoggingVCO.h"



@implementation LoggingVCO

+ (id) doCommand:(NSArray*) parameters{
    NSLog(@"JavaScriptMessage: %@",
              [parameters objectAtIndex:1]);
    return nil;
}

@end
```

For this logging to occur, a mapping must be generated between the `logMessage` command and the `LoggingVCO` class. As in the JavaScript implementation, this is done by adding `logMessage` as a key and the name of the `LoggingVCO` class as a value to a map.

Mapping is done in the QCCommandMappings.m file. The code that follows comes from this file in the DeviceCatalog example and maps `logMessage` to the `LoggingVCO` class.

```
 [aController mapCommandToVCO:@"logMessage"
      withFunction:@"LoggingVCO"];
```

The application controller is passed the `mapCommandToVCO:withFunction` message where the command is the first parameter and the VCO name is the second. This method and others like it used to map the other command object types are façades. Each of these façade methods calls the underlying `mapCommandToCO` method.

This `mapCommandToCO` method enables multiple command objects to be mapped to a single command by mapping the command to an `NSMutableArray`. This array is then used to contain the `Class` objects that match the class name passed in as the second parameter. The following code shows the implementation of the `mapCommandToCO method`.

```
- (void) mapCommandToCO:(NSString*)aCommand
           withFunction:(NSString*)aClassName
           toMap:(NSMutableDictionary*)aMap{
    NSMutableArray *controlObjects =
           [[aMap objectForKey:aCommand] retain];
    if(controlObjects == nil){
        NSMutableArray *tmpCntrlObjs =
             [[NSMutableArray alloc] initWithCapacity:1];
         [aMap setObject: tmpCntrlObjs forKey:aCommand];
```

```
            controlObjects = tmpCntrlObjs;
            [tmpCntrlObjs release];
        }
        //Get the control object's class
        //for the given name and add an object
        //of that type to the array for the command.
        Class aClass = NSClassFromString(aClassName);
        if(aClass != nil){
           [controlObjects addObject:aClass];
        }
        else{
            MESSAGE( unable to find the %@ class.
              Make sure that it exists under this
              name and try again.");
        }
}
```

The addition of the Class objects to an `NSMutableArray` enables any number of command objects of similar type, VCOs, BCOs, or others to be mapped to the same command and then executed individually in the order that the `mapCommandTo` messages were sent. Thus, you can have several VCOs execute sequentially.

For example, you can use a VCO that displays a UIView followed by another that changes another UIView's opacity, and then follow that up by logging a message. Sending three `mapCommandToVCO` messages with the same command but three different command object names would do this.

Several other examples of BCOs and VCOs exist in the DeviceCatalog example. Each one is activated as requests are made from the JavaScript portion of the application.

Summary

This chapter showed you how to activate several desirable features of iPhone or iPod Touch devices from within your JavaScript application. Using features such as GPS location, the accelerometer values, vibrating the phone, and playing sounds and audio increase the richness of your application.

By looking at the examples included in the DeviceCatalog and if you work in Objective-C, you should be able to add additional features such as scanning the Bonjour network for nearby devices, adding, removing, and retrieving contacts from the contacts application, or adding, removing, and retrieving other built in behaviors that are available in Objective-C applications.

Your JavaScript application can, using the approach described in this chapter, do most anything a pure Objective-C application can do. An example of this is in Chapter 8, where you learn how to embed Google maps into any application without losing the look and feel of Apple's Map application.

Index

A

B

C

D

E

F

G

H

I

J-K-L

L

M

N

O

P

Q

R

S

T

U

V

W

X-Y-Z