Core Bluetooth Programming Guide

Developer

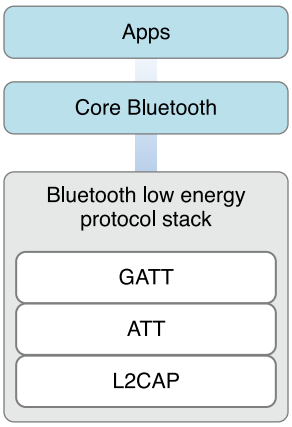
Contents

Figures

SwiftObjective-C

About Core Bluetooth

The Core Bluetooth framework provides the classes needed for your iOS and Mac apps to communicate with devices that are equipped with Bluetooth low energy wireless technology. For example, your app can discover, explore, and interact with low energy peripheral devices, such as heart rate monitors and digital thermostats. As of OS X v10.9 and iOS 6, Mac and iOS devices can also function as Bluetooth low energy peripherals, serving data to other devices, including other Mac and iOS devices.



At a Glance

Bluetooth low energy wireless technology is based on the Bluetooth 4.0 specification, which, among other things, defines a set of protocols for communicating between low energy devices. The Core Bluetooth framework is an abstraction of the Bluetooth low energy protocol stack. That said, it hides many of the low-level details of the specification from you, the developer, making it much easier for you to develop apps that interact with Bluetooth low energy devices.

Centrals and Peripherals Are the Key Players in Core Bluetooth

In Bluetooth low energy communication, there are two key players: the central and the peripheral. Each player has a different role to play in Bluetooth low energy communication. A peripheral typically has data that is needed by other devices. A central typically uses the information served up by a peripheral to accomplish some task. For example, a digital thermostat equipped with Bluetooth low energy technology might provide the temperature of a room to an iOS app that then displays the temperature in a user-friendly way.

Just as each player has a different role to play in Bluetooth low energy communication, each player performs a different set of tasks. Peripherals make their presence known by advertising the data they have over the air. Centrals, on the other hand, can scan for peripherals that might have data they're interested in. When a central discovers such a peripheral, the central can request to connect with the peripheral and begin exploring and interacting with the peripheral's data. The peripheral is then responsible for responding to the central in appropriate ways.

Relevant Chapters: Core Bluetooth Overview (page 10)

Core Bluetooth Simplifies Common Bluetooth Tasks

The Core Bluetooth framework abstracts away the low-level details from the Bluetooth 4.0 specification. As a result, many of the common Bluetooth low energy tasks you need to implement in your app are simplified. If you are developing an app that implements the central role, Core Bluetooth makes it easy to discover and connect with a peripheral, and to explore and interact with the peripheral's data. In addition, Core Bluetooth makes it easy to set up your local device to implement the peripheral role.

Relevant Chapters: Performing Common Central Role Tasks (page 17), Performing Common Peripheral Role Tasks (page 25)

iOS App States Affect Bluetooth Behavior

When your iOS app is in the background or in a suspended state, its Bluetooth-related capabilities are affected. By default, your app is unable to perform Bluetooth low energy tasks while it is in the background or in a suspended state. That said, if your app needs to perform Bluetooth low energy tasks while in the background, you can declare it to support one or both of the Core Bluetooth background execution modes (there's one for the central role, and one for the peripheral role). Even when you declare one or both of these background execution modes, certain Bluetooth tasks operate differently while your app is in the background. You want to take these differences into account when designing your app.

Even apps that support background processing may be terminated by the system at any time to free up memory for the current foreground app. As of iOS 7, Core Bluetooth supports saving state information for central and peripheral manager objects and restoring that state at app launch time. You can use this feature to support long-term actions involving Bluetooth devices.

Relevant Chapters: Core Bluetooth Background Processing for iOS Apps (page 35)

Follow Best Practices to Enhance the User Experience

The Core Bluetooth framework gives your app control over many of the common Bluetooth low energy transactions. Follow best practices to harness this level of control in a responsible way and enhance the user's experience.

For example, many of the tasks you perform when implementing the central or the peripheral role use your device's onboard radio to transmit signals over the air. Because your device's radio is shared with other forms of wireless communication, and because radio usage has an adverse effect on a device's battery life, always design your app to minimize how much it uses the radio.

Relevant Chapters: Best Practices for Interacting with a Remote Peripheral Device (page 45), Best Practices for Setting Up Your Local Device as a Peripheral (page 51)

How to Use This Document

If you have never used the Core Bluetooth framework, or if you are unfamiliar with basic Bluetooth low energy concepts, read this document in its entirety. In Core Bluetooth Overview (page 10), you learn the key terms and concepts that you need to know for the remainder of the book.

After you understand the key concepts, read Performing Common Central Role Tasks (page 17) to learn how to develop your app to implement the central role on your local device. Similarly, to learn how to develop your app to implement the peripheral role on your local device, read Performing Common Peripheral Role Tasks (page 25).

To ensure that your app is performing well and adhering to best practices, read the later chapters: Core Bluetooth Background Processing for iOS Apps (page 35), Best Practices for Interacting with a Remote Peripheral Device (page 45), and Best Practices for Setting Up Your Local Device as a Peripheral (page 51).

See Also

The official Bluetooth Special Interest Group (SIG) website provides the definitive information about Bluetooth low energy wireless technology. There, you can also find the Bluetooth 4.0 specification.

If you are designing hardware accessories that use Bluetooth low energy technology to communicate with Apple products, including Mac, iPhone, iPad, and iPod touch models, read Bluetooth Accessory Design Guidelines for Apple Products. If your Bluetooth accessory (that connects to an iOS device through a Bluetooth low energy link) needs access to notifications that are generated on iOS devices, read *Apple Notification Center Service (ANCS) Specification*.

Core Bluetooth Overview

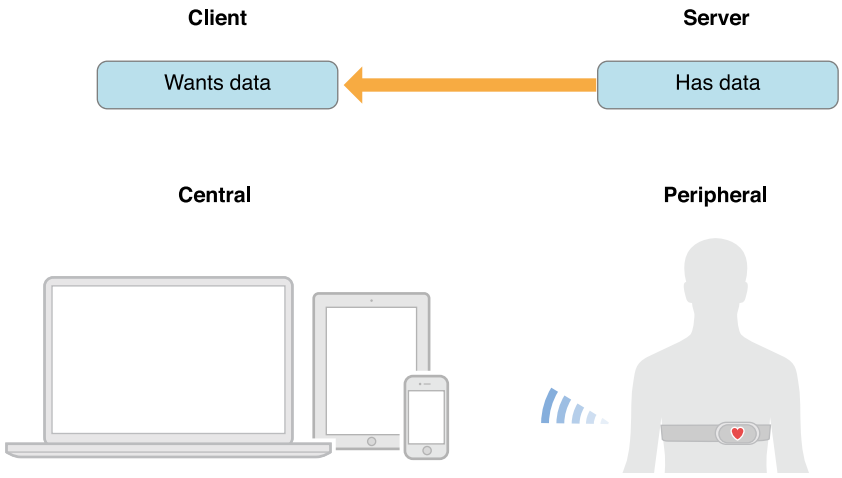
The Core Bluetooth framework lets your iOS and Mac apps communicate with Bluetooth low energy devices. For example, your app can discover, explore, and interact with low energy peripheral devices, such as heart rate monitors, digital thermostats, and even other iOS devices.

The framework is an abstraction of the Bluetooth 4.0 specification for use with low energy devices. That said, it hides many of the low-level details of the specification from you, the developer, making it much easier for you to develop apps that interact with Bluetooth low energy devices. Because the framework is based on the specification, some concepts and terminology from the specification have been adopted. This chapter introduces you to the key terms and concepts that you need to know to begin developing great apps using the Core Bluetooth framework.

Central and Peripheral Devices and Their Roles in Bluetooth Communication

There are two major players involved in all Bluetooth low energy communication: the central and the peripheral. Based on a somewhat traditional client-server architecture, a *peripheral* typically has data that is needed by other devices. A *central* typically uses the information served up by peripherals to accomplish some particular task. As Figure 1-1 shows, for example, a heart rate monitor may have useful information that your Mac or iOS app may need in order to display the user's heart rate in a user-friendly way.

Figure 1-1 Central and peripheral devices



Centrals Discover and Connect to Peripherals That Are Advertising

Peripherals broadcast some of the data they have in the form of advertising packets. An *advertising packet* is a relatively small bundle of data that may contain useful information about what a peripheral has to offer, such as the peripheral's name and primary functionality. For instance, a digital thermostat may advertise that it provides the current temperature of a room. In Bluetooth low energy, advertising is the primary way that peripherals make their presence known.

A central, on the other hand, can scan and listen for any peripheral device that is advertising information that it's interested in, as shown in Figure 1-2. A central can ask to connect to any peripheral that it has discovered advertising.

Figure 1-2 Advertising and discovery

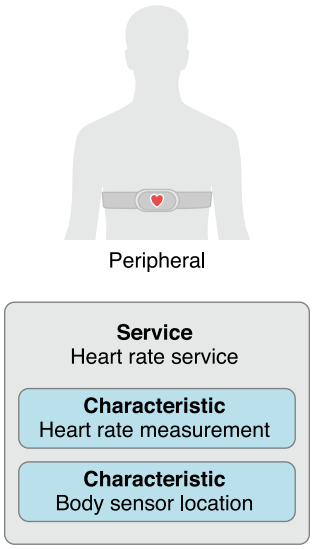
How the Data of a Peripheral Is Structured

The purpose of connecting to a peripheral is to begin exploring and interacting with the data it has to offer. Before you can do this, however, it helps to understand how the data of a peripheral is structured.

Peripherals may contain one or more services or provide useful information about their connected signal strength. A *service* is a collection of data and associated behaviors for accomplishing a function or feature of a device (or portions of that device). For example, one service of a heart rate monitor may be to expose heart rate data from the monitor's heart rate sensor.

Services themselves are made up of characteristics or included services (that is, references to other services). A *characteristic* provides further details about a peripheral's service. For example, the heart rate service just described may contain one characteristic that describes the intended body location of the device's heart rate sensor and another characteristic that transmits heart rate measurement data. Figure 1-3 illustrates one possible structure of a heart rate monitor's service and characteristics.

Figure 1-3 A peripheral's service and characteristics



Centrals Explore and Interact with the Data on a Peripheral

After a central has successfully established a connection to a peripheral, it can discover the full range of services and characteristics the peripheral has to offer (advertising data may contain only a fraction of the available services).

A central can also interact with a peripheral's service by reading or writing the value of that service's characteristic. For example, your app may request the current room temperature from a digital thermostat, or it may provide the thermostat with a value at which to set the room's temperature.

How Centrals, Peripherals, and Peripheral Data Are Represented

The major players and data involved in Bluetooth low energy communication are mapped onto the Core Bluetooth framework in a simple, straightforward way.

Objects on the Central Side

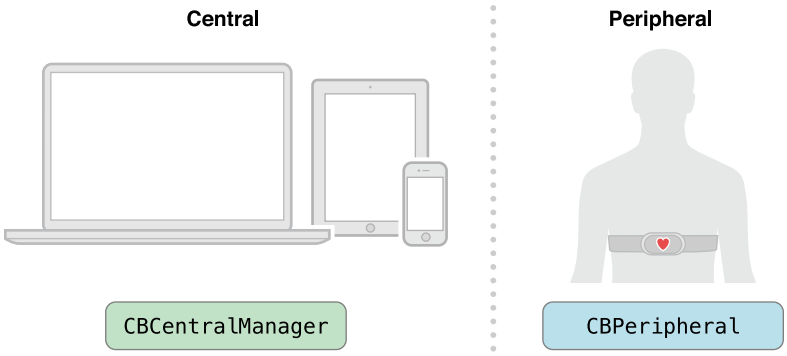
When you are using a local central to interact with a remote peripheral, you are performing actions on the central side of Bluetooth low energy communication. Unless you are setting up a local peripheral device—and using it to respond to requests by a central—most of your Bluetooth transactions will take place on the central side.

For information about how to implement the central role in your app, see Performing Common Central Role Tasks (page 17) and Best Practices for Interacting with a Remote Peripheral Device (page 45)

Local Centrals and Remote Peripherals

On the central side, a local central device is represented by a `CBCentralManager` object. These objects are used to manage discovered or connected remote peripheral devices (represented by `CBPeripheral` objects), including scanning for, discovering, and connecting to advertising peripherals. Figure 1-4 shows how local centrals and remote peripherals are represented in the Core Bluetooth framework.

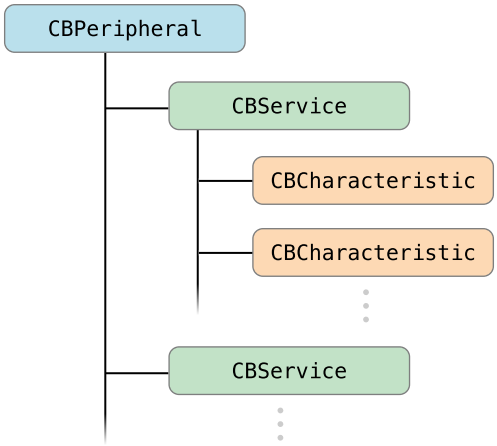
Figure 1-4 Core Bluetooth objects on the central side



A Remote Peripheral's Data Are Represented by CBService and CBCharacteristic Objects

When you are interacting with the data on a remote peripheral (represented by a `CBPeripheral` object), you are dealing with its services and characteristics. In the Core Bluetooth framework, the services of a remote peripheral are represented by `CBService` objects. Similarly, the characteristics of a remote peripheral's service are represented by `CBCharacteristic` objects. Figure 1-5 illustrates the basic structure of a remote peripheral's services and characteristics.

Figure 1-5 A remote peripheral's tree of services and characteristics



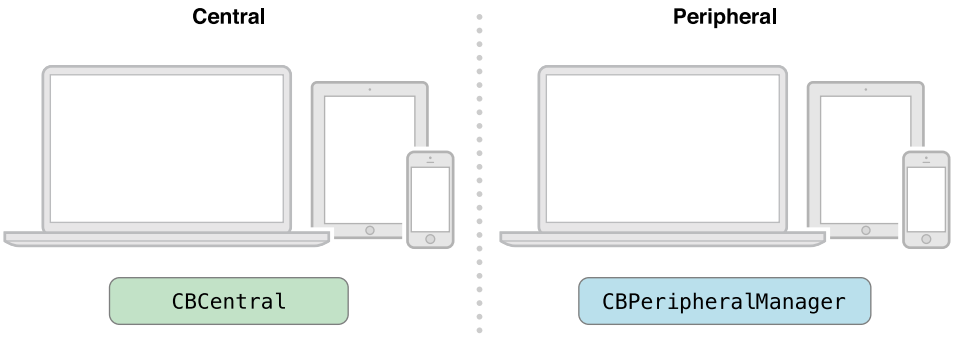
Objects on the Peripheral Side

As of OS X v10.9 and iOS 6, Mac and iOS devices can function as Bluetooth low energy peripherals, serving data to other devices, including other Macs, iPhones, and iPads. When setting up your device to implement the peripheral role, you are performing actions on the peripheral side of Bluetooth low energy communication.

Local Peripherals and Remote Centrals

On the peripheral side, a local peripheral device is represented by a `CBPeripheralManager` object. These objects are used to manage published services within the local peripheral device's database of services and characteristics and to advertise these services to remote central devices (represented by `CBCentral` objects). Peripheral manager objects are also used to respond to read and write requests from these remote centrals. Figure 1-6 shows how local peripherals and remote centrals are represented in the Core Bluetooth framework.

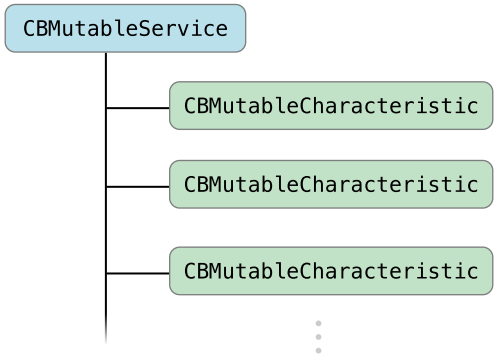
Figure 1-6 Core Bluetooth objects on the peripheral side



A Local Peripheral's Data Are Represented by CBMutableService and CBMutableCharacteristic Objects

When you are setting up and interacting with the data on a local peripheral (represented by a `CBPeripheralManager` object), you are dealing with mutable versions of its services and characteristics. In the Core Bluetooth framework, the services of a local peripheral are represented by `CBMutableService` objects. Similarly, the characteristics of a local peripheral's service are represented by `CBMutableCharacteristic` objects. Figure 1-7 illustrates the basic structure of a local peripheral's services and characteristics.

Figure 1-7 A local peripheral's tree of services and characteristics



For more information about how to set up your local device to implement the peripheral role, see Performing Common Peripheral Role Tasks (page 25) and Best Practices for Setting Up Your Local Device as a Peripheral (page 51).

Performing Common Central Role Tasks

SwiftObjective-C

Devices that implement the central role in Bluetooth low energy communication perform a number of common tasks—for example, discovering and connecting to available peripherals, and exploring and interacting with the data that peripherals have to offer. In contrast, devices that implement the peripheral role also perform a number of common, but different, tasks—for example, publishing and advertising services, and responding to read, write, and subscription requests from connected centrals.

In this chapter, you will learn how to use the Core Bluetooth framework to perform the most common types of Bluetooth low energy tasks from the central side. The code-based examples that follow will assist you in developing your app to implement the central role on your local device. Specifically, you will learn how to:

- Start up a central manager object
- Discover and connect to peripheral devices that are advertising
- Explore the data on a peripheral device after you've connected to it
- Send read and write requests to a characteristic value of a peripheral's service
- Subscribe to a characteristic's value to be notified when it is updated

In the next chapter, you will learn how to develop your app to implement the peripheral role on your local device.

The code examples that you find in this chapter are simple and abstract; you may need to make appropriate changes to incorporate them into your real world app. More advanced topics related to implementing the central role—including tips, tricks, and best practices—are covered in the later chapters, Core Bluetooth Background Processing for iOS Apps (page 35) and Best Practices for Interacting with a Remote Peripheral Device (page 45).

Starting Up a Central Manager

Since a `CBCentralManager` object is the Core Bluetooth object-oriented representation of a local central device, you must allocate and initialize a central manager instance before you can perform any Bluetooth low energy transactions. You can start up your central manager by calling the `initWithDelegate:queue:options:` method of the `CBCentralManager` class, like this:

```
myCentralManager =
    [[CBCentralManager alloc] initWithDelegate:self queue:nil options:nil];
```

In this example, `self` is set as the delegate to receive any central role events. By specifying the dispatch queue as `nil`, the central manager dispatches central role events using the main queue.

When you create a central manager, the central manager calls the `centralManagerDidUpdateState:` method of its delegate object. You must implement this delegate method to ensure that Bluetooth low energy is supported and available to use on the central device. For more information about how to implement this delegate method, see *CBCentralManagerDelegate Protocol Reference*.

Discovering Peripheral Devices That Are Advertising

One of the first central-side tasks that you are likely to perform is to discover what peripheral devices are available for your app to connect to. As mentioned earlier in Centrals Discover and Connect to Peripherals That Are Advertising (page 11), advertising is the primary way that peripherals make their presence known. You can discover any peripheral devices that are advertising by calling the `scanForPeripheralsWithServices:options:` method of the `CBCentralManager` class, like this:

```
[myCentralManager scanForPeripheralsWithServices:nil options:nil];
```

> **Note:** If you specify `nil` for the first parameter, the central manager returns *all* discovered peripherals, regardless of their supported services. In a real app, you will likely specify an array of `CBUUID` objects, each of which represents the universally unique identifier (UUID) of a service that a peripheral is advertising. When you specify an array of service UUIDs, the central manager returns only peripherals that advertise those services, allowing you to scan only for devices that you may be interested in.
>
> UUIDs, and the `CBUUID` objects that represent them, are discussed in more detail in Services and Characteristics Are Identified by UUIDs (page 26).

After you call the `scanForPeripheralsWithServices:options:` method to discover what peripherals are available, the central manager calls the `centralManager:didDiscoverPeripheral:advertisementData:RSSI:` method of its delegate object each time a peripheral is discovered. Any peripheral that is discovered is returned as a `CBPeripheral` object. As the following shows, you can implement this delegate method to list any peripheral that is discovered:

```
- (void)centralManager:(CBCentralManager *)central
```

```
didDiscoverPeripheral:(CBPeripheral *)peripheral
    advertisementData:(NSDictionary *)advertisementData
                 RSSI:(NSNumber *)RSSI {

   NSLog(@"Discovered %@", peripheral.name);
   ...
```

When you have found a peripheral device that you're interested in connecting to, stop scanning for other devices in order to save power.

```
    [myCentralManager stopScan];
    NSLog(@"Scanning stopped");
```

Connecting to a Peripheral Device After You've Discovered It

After you have discovered a peripheral device that is advertising services you are interested in, you can request a connection to the peripheral by calling the `connectPeripheral:options:` method of the `CBCentralManager` class. Simply call this method and specify the discovered peripheral that you want to connect to, like this:

```
    [myCentralManager connectPeripheral:peripheral options:nil];
```

Assuming that the connection request is successful, the central manager calls the `centralManager:didConnectPeripheral:` method of its delegate object, which you can implement to log that the connection is established, as the following shows:

```
- (void)centralManager:(CBCentralManager *)central
  didConnectPeripheral:(CBPeripheral *)peripheral {


    NSLog(@"Peripheral connected");
    ...
```

Before you begin interacting with the peripheral, you should set the peripheral's delegate to ensure that it receives the appropriate callbacks, like this:

```
    peripheral.delegate = self;
```

Discovering the Services of a Peripheral That You're Connected To

After you have established a connection to a peripheral, you can begin to explore its data. The first step in exploring what a peripheral has to offer is discovering its available services. Because there are size restrictions on the amount of data a peripheral can advertise, you may discover that a peripheral has more services than what it advertises (in its advertising packets). You can discover all of the services that a peripheral offers by calling the `discoverServices:` method of the `CBPeripheral` class, like this:

```
    [peripheral discoverServices:nil];
```

> **Note:** In a real app, you will not likely pass in `nil` as the parameter, since doing so returns *all* the services available on a peripheral device. Because a peripheral may contain many more services than you are interested in, discovering all of them may waste battery life and be an unnecessary use of time. More likely, you will specify the UUIDs of the services that you already know you are interested in discovering, as shown in Explore a Peripheral's Data Wisely (page 46).

When the specified services are discovered, the peripheral (the `CBPeripheral` object you're connected to) calls the `peripheral:didDiscoverServices:` method of its delegate object. Core Bluetooth creates an array of `CBService` objects—one for each service that is discovered on the peripheral. As the following shows, you can implement this delegate method to access the array of discovered services:

```
- (void)peripheral:(CBPeripheral *)peripheral
didDiscoverServices:(NSError *)error {

    for (CBService *service in peripheral.services) {
        NSLog(@"Discovered service %@", service);
        ...
    }
    ...
```

Discovering the Characteristics of a Service

Assuming that you have found a service that you are interested in, the next step in exploring what a peripheral has to offer is discovering all of the service's characteristics. Discovering all of the characteristics of a service is as simple as calling the `discoverCharacteristics:forService:` method of the `CBPeripheral` class, specifying the appropriate service, like this:

```
NSLog(@"Discovering characteristics for service %@", interestingService);
[peripheral discoverCharacteristics:nil forService:interestingService];
```

> **Note:** In a real app, you will not likely pass in `nil` as the first parameter, since doing so returns *all* the characteristics of a peripheral's service. Because a peripheral's service may contain many more characteristics than you are interested in, discovering all of them may waste battery life and be an unnecessary use of time. More likely, you will specify the UUIDs of the characteristics that you already know you are interested in discovering.

The peripheral calls the `peripheral:didDiscoverCharacteristicsForService:error:` method of its delegate object when the characteristics of the specified service are discovered. Core Bluetooth creates an array of `CBCharacteristic` objects—one for each characteristic that is discovered. The following example shows how you can implement this delegate method to simply log every characteristic that is discovered:

```
- (void)peripheral:(CBPeripheral *)peripheral
didDiscoverCharacteristicsForService:(CBService *)service
             error:(NSError *)error {

    for (CBCharacteristic *characteristic in service.characteristics) {
        NSLog(@"Discovered characteristic %@", characteristic);
        ...
    }
    ...
```

Retrieving the Value of a Characteristic

A characteristic contains a single value that represents more information about a peripheral's service. For example, a temperature measurement characteristic of a health thermometer service may have a value that indicates a temperature in Celsius. You can retrieve the value of a characteristic by reading it directly or by subscribing to it.

Reading the Value of a Characteristic

After you have found a characteristic of a service that you are interested in, you can read the characteristic's value by calling the `readValueForCharacteristic:` method of the `CBPeripheral` class, specifying the appropriate characteristic, like this:

```
NSLog(@"Reading value for characteristic %@", interestingCharacteristic);
[peripheral readValueForCharacteristic:interestingCharacteristic];
```

When you attempt to read the value of a characteristic, the peripheral calls the `peripheral:didUpdateValueForCharacteristic:error:` method of its delegate object to retrieve the value. If the value is successfully retrieved, you can access it through the characteristic's value property, like this:

```
- (void)peripheral:(CBPeripheral *)peripheral
didUpdateValueForCharacteristic:(CBCharacteristic *)characteristic
             error:(NSError *)error {

    NSData *data = characteristic.value;
    // parse the data as needed
    ...
```

> **Note:** Not all characteristics are guaranteed to have a value that is readable. You can determine whether a characteristic's value is readable by accessing the `CBCharacteristicPropertyRead` constant, detailed in *CBCharacteristic Class Reference*. If you try to read a value that is not readable, the `peripheral:didUpdateValueForCharacteristic:error:` delegate method returns a suitable error.

Subscribing to a Characteristic's Value

Though reading the value of a characteristic using the `readValueForCharacteristic:` method can be effective for some use cases, it is not the most efficient way to retrieve a value that changes. For most characteristic values that change—for instance, your heart rate at any given time—you should retrieve them by subscribing to them. When you subscribe to a characteristic's value, you receive a notification from the peripheral when the value changes.

You can subscribe to the value of a characteristic that you are interested in by calling the `setNotifyValue:forCharacteristic:` method of the `CBPeripheral` class, specifying the first parameter as `YES`, like this:

```
[peripheral setNotifyValue:YES forCharacteristic:interestingCharacteristic];
```

When you attempt to subscribe (or unsubscribe) to a characteristic's value, the peripheral calls the `peripheral:didUpdateNotificationStateForCharacteristic:error:` method of its delegate object. If the subscription request fails for any reason, you can implement this delegate method to access the cause of the error, as the following example shows:

```
- (void)peripheral:(CBPeripheral *)peripheral
didUpdateNotificationStateForCharacteristic:(CBCharacteristic *)characteristic
             error:(NSError *)error {

    if (error) {
        NSLog(@"Error changing notification state: %@",
            [error localizedDescription]);
    }
    ...
```

Note: Not all characteristics are configured to allow you to subscribe to their values. You can determine whether a characteristic is so configured by accessing the relevant constants in the `Characteristic Properties` enumeration, detailed in *CBCharacteristic Class Reference*.

After you have successfully subscribed to a characteristic's value, the peripheral device notifies your app when the value has changed. Each time the value changes, the peripheral calls the `peripheral:didUpdateValueForCharacteristic:error:` method of its delegate object. To retrieve the updated value, you can implement this method in the same way as described above in Reading the Value of a Characteristic (page 22).

Writing the Value of a Characteristic

For some use cases, it makes sense to write the value of a characteristic. For example, if your app interacts with a Bluetooth low energy digital thermostat, you may want to provide the thermostat with a value at which to set the room's temperature. If a characteristic's value is writeable, you can write its value with some data (an instance of `NSData`) by calling the `writeValue:forCharacteristic:type:` method of the `CBPeripheral` class, like this:

```
NSLog(@"Writing value for characteristic %@", interestingCharacteristic);
[peripheral writeValue:dataToWrite forCharacteristic:interestingCharacteristic
    type:CBCharacteristicWriteWithResponse];
```

When you attempt to write the value of a characteristic, you specify what type of write you want to perform. In the example above, the write type is specified as `CBCharacteristicWriteWithResponse`, which indicates that the peripheral lets your app know whether the write is successful. For more information about the write types that are supported in the Core Bluetooth framework, see the `CBCharacteristicWriteType` enumeration in *CBPeripheral Class Reference*.

The peripheral responds to write requests that are specified as `CBCharacteristicWriteWithResponse` by calling the `peripheral:didWriteValueForCharacteristic:error:` method of its delegate object. If the write fails for any reason, you can implement this delegate method to access the cause of the error, as the following example shows:

```
- (void)peripheral:(CBPeripheral *)peripheral
didWriteValueForCharacteristic:(CBCharacteristic *)characteristic
             error:(NSError *)error {

    if (error) {
        NSLog(@"Error writing characteristic value: %@",
            [error localizedDescription]);
    }
    ...
```

> **Note:** Characteristics may only allow certain types of writes to be performed on their value. To determine which types of writes are permitted to a characteristic's value, you access the relevant properties of the `Characteristic Properties` enumeration, detailed in *CBCharacteristic Class Reference*.

Performing Common Peripheral Role Tasks

SwiftObjective-C

In the last chapter, you learned how to perform the most common types of Bluetooth low energy tasks from the central side. In this chapter, you learn how to use the Core Bluetooth framework to perform the most common types of Bluetooth low energy tasks from the peripheral side. The code-based examples that follow will assist you in developing your app to implement the peripheral role on your local device. Specifically, you will learn how to:

- Start up a peripheral manager object
- Set up services and characteristics on your local peripheral
- Publish your services and characteristics to your device's local database
- Advertise your services
- Respond to read and write requests from a connected central
- Send updated characteristic values to subscribed centrals

The code examples that you find in this chapter are simple and abstract; you may need to make appropriate changes to incorporate them into your real-world app. More advanced topics related to implementing the peripheral role on your local device—including tips, tricks, and best practices—are covered in the later chapters, Core Bluetooth Background Processing for iOS Apps (page 35) and Best Practices for Setting Up Your Local Device as a Peripheral (page 51).

Starting Up a Peripheral Manager

The first step in implementing the peripheral role on your local device is to allocate and initialize a peripheral manager instance (represented by a `CBPeripheralManager` object). Start up your peripheral manager by calling the `initWithDelegate:queue:options:` method of the `CBPeripheralManager` class, like this:

```
myPeripheralManager =
    [[CBPeripheralManager alloc] initWithDelegate:self queue:nil options:nil];
```

In this example, `self` is set as the delegate to receive any peripheral role events. When you specify the dispatch queue as `nil`, the peripheral manager dispatches peripheral role events using the main queue.

When you create a peripheral manager, the peripheral manager calls the `peripheralManagerDidUpdateState:` method of its delegate object. You must implement this delegate method to ensure that Bluetooth low energy is supported and available to use on the local peripheral device. For more information about how to implement this delegate method, see *CBPeripheralManagerDelegate Protocol Reference*.

Setting Up Your Services and Characteristics

As shown in Figure 1-7 (page 15), a local peripheral's database of services and characteristics is organized in a tree-like manner. You must organize them in this tree-like manner to set up your services and characteristics on your local peripheral. Your first step in carrying out these tasks is understanding how services and characteristics are identified.

Services and Characteristics Are Identified by UUIDs

The services and characteristics of a peripheral are identified by 128-bit Bluetooth-specific UUIDs, which are represented in the Core Bluetooth framework by `CBUUID` objects. Though not all UUIDs that identify a service or characteristic are predefined by the Bluetooth Special Interest Group (SIG), Bluetooth SIG has defined and published a number of commonly used UUIDs that have been shortened to 16-bits for convenience. For example, Bluetooth SIG has predefined the 16-bit UUID that identifies a heart rate service as 180D. This UUID is shortened from its equivalent 128-bit UUID, 0000180D-0000-1000-8000-00805F9B34FB, which is based on the Bluetooth base UUID that is defined in the Bluetooth 4.0 specification, Volume 3, Part F, Section 3.2.1.

The `CBUUID` class provides factory methods that make it much easier to deal with long UUIDs when developing your app. For example, instead of passing around the string representation of the heart rate service's 128-bit UUID in your code, you can simply use the `UUIDWithString` method to create a `CBUUID` object from the service's predefined 16-bit UUID, like this:

```
CBUUID *heartRateServiceUUID = [CBUUID UUIDWithString: @"180D"];
```

When you create a `CBUUID` object from a predefined 16-bit UUID, Core Bluetooth prefills the rest of 128-bit UUID with the Bluetooth base UUID.

Create Your Own UUIDs for Custom Services and Characteristics

You may have services and characteristics that are not identified by predefined Bluetooth UUIDs. If you do, you need to generate your own 128-bit UUIDs to identify them.

Use the command-line utility `uuidgen` to easily generate 128-bit UUIDs. To get started, open a window in Terminal. Next, for each service and characteristic that you need to identify with a UUID, type `uuidgen` on the command line to receive a unique 128-bit value in the form of an ASCII string that is punctuated by hyphens, as in the following example:

```
$ uuidgen
71DA3FD1-7E10-41C1-B16F-4430B506CDE7
```

You can then use this UUID to create a `CBUUID` object using the `UUIDWithString` method, like this:

```
    CBUUID *myCustomServiceUUID =
        [CBUUID UUIDWithString:@"71DA3FD1-7E10-41C1-B16F-4430B506CDE7"];
```

Build Your Tree of Services and Characteristics

After you have the UUIDs of your services and characteristics (represented by `CBUUID` objects), you can create mutable services and characteristics and organize them in the tree-like manner described above. For example, if you have the UUID of a characteristic, you can create a mutable characteristic by calling the `initWithType:properties:value:permissions:` method of the `CBMutableCharacteristic` class, like this:

```
    myCharacteristic =
        [[CBMutableCharacteristic alloc] initWithType:myCharacteristicUUID
         properties:CBCharacteristicPropertyRead
         value:myValue permissions:CBAttributePermissionsReadable];
```

When you create a mutable characteristic, you set its properties, value, and permissions. The properties and permissions you set determine, among other things, whether the value of the characteristic is readable or writeable, and whether a connected central can subscribe to the characteristic's value. In this example, the value of the characteristic is set to be readable by a connected central. For more information about the range of supported properties and permissions of mutable characteristics, see *CBMutableCharacteristic Class Reference* .

> **Note:** If you specify a value for the characteristic, the value is cached and its properties and permissions are set to be readable. Therefore, if you need the value of a characteristic to be writeable, or if you expect the value to change during the lifetime of the published service to which the characteristic belongs, you must specify the value to be `nil`. Following this approach ensures that the value is treated dynamically and requested by the peripheral manager whenever the peripheral manager receives a read or write request from a connected central.

Now that you have created a mutable characteristic, you can create a mutable service to associate the characteristic with. To do so, call the `initWithType:primary:` method of the `CBMutableService` class, as shown here:

```
myService = [[CBMutableService alloc] initWithType:myServiceUUID primary:YES];
```

In this example, the second parameter is set to `YES`, indicating that the service is primary as opposed to secondary. A *primary service* describes the primary functionality of a device and can be included (referenced) by another service. A *secondary service* describes a service that is relevant only in the context of another service that has referenced it. For example, the primary service of a heart rate monitor may be to expose heart rate data from the monitor's heart rate sensor, whereas a secondary service may be to expose the sensor's battery data.

After you create a service, you can associate the characteristic with it by setting the service's array of characteristics, like this:

```
myService.characteristics = @[myCharacteristic];
```

Publishing Your Services and Characteristics

After you have built your tree of services and characteristics, the next step in implementing the peripheral role on your local device is publishing them to the device's database of services and characteristics. This task is easy to perform using the Core Bluetooth framework. You call the `addService:` method of the `CBPeripheralManager` class, like this:

```
[myPeripheralManager addService:myService];
```

When you call this method to publish your services, the peripheral manager calls the `peripheralManager:didAddService:error:` method of its delegate object. If an error occurs and your services can't be published, implement this delegate method to access the cause of the error, as the following example shows:

```
- (void)peripheralManager:(CBPeripheralManager *)peripheral
            didAddService:(CBService *)service
                    error:(NSError *)error {

    if (error) {
        NSLog(@"Error publishing service: %@", [error localizedDescription]);
    }
    ...
```

Note: After you publish a service and any of its associated characteristics to the peripheral's database, the service is cached and you can no longer make changes to it.

Advertising Your Services

When you have published your services and characteristics to your device's database of services and characteristics, you are ready to start advertising some of them to any centrals that may be listening. As the following example shows, you can advertise some of your services by calling the `startAdvertising:` method of the `CBPeripheralManager` class, passing in a dictionary (an instance of `NSDictionary`) of advertisement data:

```
    [myPeripheralManager startAdvertising:@{ CBAdvertisementDataServiceUUIDsKey :
        @[myFirstService.UUID, mySecondService.UUID] }];
```

In this example, the only key in the dictionary, `CBAdvertisementDataServiceUUIDsKey`, expects as a value an array (an instance of `NSArray`) of `CBUUID` objects that represent the UUIDs of the services you want to advertise. The possible keys that you may specify in a dictionary of advertisement data are detailed in the constants described in `Advertisement Data Retrieval Keys` in *CBCentralManagerDelegate Protocol Reference*. That said, only two of the keys are supported for peripheral manager objects: `CBAdvertisementDataLocalNameKey` and `CBAdvertisementDataServiceUUIDsKey`.

When you start advertising some of the data on your local peripheral, the peripheral manager calls the `peripheralManagerDidStartAdvertising:error:` method of its delegate object. If an error occurs and your services can't be advertised, implement this delegate method to access the cause of the error, like this:

```
- (void)peripheralManagerDidStartAdvertising:(CBPeripheralManager *)peripheral
                                       error:(NSError *)error {

    if (error) {
        NSLog(@"Error advertising: %@", [error localizedDescription]);
    }
    ...
```

> **Note:** Data advertising is done on a "best effort" basis, because space is limited and there may be multiple apps advertising simultaneously. For more information, see the discussion of the `startAdvertising:` method in *CBPeripheralManager Class Reference*.
>
> Advertising behavior is also affected when your app is in the background. This topic is discussed in the next chapter, Core Bluetooth Background Processing for iOS Apps (page 35).

Once you begin advertising data, remote centrals can discover and initiate a connection with you.

Responding to Read and Write Requests from a Central

After you are connected to one or more remote centrals, you may begin receiving read or write requests from them. When you do, be sure to respond to those requests in an appropriate manner. The following examples describe how to handle such requests.

When a connected central requests to read the value of one of your characteristics, the peripheral manager calls the `peripheralManager:didReceiveReadRequest:` method of its delegate object. The delegate method delivers the request to you in the form of a `CBATTRequest` object, which has a number of properties that you can use to fulfill the request.

For example, when you receive a simple request to read the value of a characteristic, the properties of the `CBATTRequest` object you receive from the delegate method can be used to make sure that the characteristic in your device's database matches the one that the remote central specified in the original read request. You can begin to implement this delegate method, like this:

```
- (void)peripheralManager:(CBPeripheralManager *)peripheral
    didReceiveReadRequest:(CBATTRequest *)request {

    if ([request.characteristic.UUID isEqual:myCharacteristic.UUID]) {
        ...
```

If the characteristics' UUIDs match, the next step is to make sure that the read request isn't asking to read from an index position that is outside the bounds of your characteristic's value. As the following example shows, you can use a `CBATTRequest` object's `offset` property to ensure the read request isn't attempting to read outside the proper bounds:

```
    if (request.offset > myCharacteristic.value.length) {
        [myPeripheralManager respondToRequest:request
            withResult:CBATTErrorInvalidOffset];
        return;
    }
```

Assuming the request's offset is verified, now set the value of the request's characteristic property (whose value by default is `nil`) to the value of the characteristic you created on your local peripheral, taking into account the offset of the read request:

```
    request.value = [myCharacteristic.value
        subdataWithRange:NSMakeRange(request.offset,
        myCharacteristic.value.length - request.offset)];
```

After you set the value, respond to the remote central to indicate that the request was successfully fulfilled. Do so by calling the `respondToRequest:withResult:` method of the `CBPeripheralManager` class, passing back the request (whose value you updated) and the result of the request, like this:

```
    [myPeripheralManager respondToRequest:request withResult:CBATTErrorSuccess];
    ...
```

Call the `respondToRequest:withResult:` method exactly once each time the `peripheralManager:didReceiveReadRequest:` delegate method is called.

> **Note:** If the characteristics' UUIDs do not match, or if the read can not be completed for any other reason, you would not attempt to fulfill the request. Instead, you would call the `respondToRequest:withResult:` method immediately and provide a result that indicated the cause of the failure. For a list of the possible results you may specify, see the `CBATTError Constants` enumeration in *Core Bluetooth Constants Reference*.

Handling write requests from a connected central is also straightforward. When a connected central sends a request to write the value of one or more of your characteristics, the peripheral manager calls the `peripheralManager:didReceiveWriteRequests:` method of its delegate object. This time, the delegate method delivers the requests to you in the form of an array containing one or more `CBATTRequest` objects, each representing a write request. After you have ensured that a write request can be fulfilled, you can write the characteristic's value, like this:

```
myCharacteristic.value = request.value;
```

Although the above example does not demonstrate this, be sure to take into account the request's offset property when writing the value of your characteristic.

Just as you respond to a read request, call the `respondToRequest:withResult:` method exactly once each time the `peripheralManager:didReceiveWriteRequests:` delegate method is called. That said, the first parameter of the `respondToRequest:withResult:` method expects a single `CBATTRequest` object, even though you may have received an array containing more than one of them from the `peripheralManager:didReceiveWriteRequests:` delegate method. You should pass in the first request of the array, like this:

```
[myPeripheralManager respondToRequest:[requests objectAtIndex:0]
    withResult:CBATTErrorSuccess];
```

Note: Treat multiple requests as you would a single request—if any individual request cannot be fulfilled, you should not fulfill any of them. Instead, call the `respondToRequest:withResult:` method immediately and provide a result that indicates the cause of the failure.

Sending Updated Characteristic Values to Subscribed Centrals

Often, connected centrals will subscribe to one or more of your characteristic values, as described in Subscribing to a Characteristic's Value (page 22). When they do, you are responsible for sending them notifications when the value of characteristic they subscribed to changes. The following examples describe how.

When a connected central subscribes to the value of one of your characteristics, the peripheral manager calls the `peripheralManager:central:didSubscribeToCharacteristic:` method of its delegate object:

```
- (void)peripheralManager:(CBPeripheralManager *)peripheral
                  central:(CBCentral *)central
didSubscribeToCharacteristic:(CBCharacteristic *)characteristic {

    NSLog(@"Central subscribed to characteristic %@", characteristic);
    ...
```

Use the above delegate method as a cue to start sending the central updated values.

Next, get the updated value of the characteristic and send it to the central by calling the `updateValue:forCharacteristic:onSubscribedCentrals:` method of the `CBPeripheralManager` class.

```
    NSData *updatedValue = // fetch the characteristic's new value
    BOOL didSendValue = [myPeripheralManager updateValue:updatedValue
        forCharacteristic:characteristic onSubscribedCentrals:nil];
```

When you call this method to send updated characteristic values to subscribed centrals, you can specify which centrals you want to update in the last parameter. As in the above example, if you specify `nil`, all connected and subscribed centrals are updated (and any connected centrals that have not subscribed are ignored).

The `updateValue:forCharacteristic:onSubscribedCentrals:` method returns a Boolean value that indicates whether the update was successfully sent to the subscribed centrals. If the underlying queue that is used to transmit the updated value is full, the method returns `NO`. The peripheral manager then calls the

`peripheralManagerIsReadyToUpdateSubscribers:` method of its delegate object when more space in the transmit queue becomes available. You can then implement this delegate method to resend the value, again using the `updateValue:forCharacteristic:onSubscribedCentrals:` method.

> **Note:** Use notifications to send a single packet of data to subscribed centrals. That is, when you update a subscribed central, you should send the entire updated value in a single notification, by calling the `updateValue:forCharacteristic:onSubscribedCentrals:` method only once.
>
> Depending on the size of your characteristic's value, not all of the data may be transmitted by the notification. If this happens, the situation should be handled on the central side through a call to the `readValueForCharacteristic:` method of the `CBPeripheral` class, which can retrieve the entire value.

Core Bluetooth Background Processing for iOS Apps

For iOS apps, it is crucial to know whether your app is running in the foreground or the background. An app must behave differently in the background than in the foreground, because system resources are more limited on iOS devices. For an overall discussion of multitasking on iOS, see App States and Multitasking in *App Programming Guide for iOS*.

By default, many of the common Core Bluetooth tasks—on both the central and peripheral side—are disabled while your app is in the background or in a suspended state. That said, you can declare your app to support the Core Bluetooth background execution modes to allow your app to be woken up from a suspended state to process certain Bluetooth-related events. Even if your app doesn't need the full range of background processing support, it can still ask to be alerted by the system when important events occur.

Even if your app supports one or both of the Core Bluetooth background execution modes, it can't run forever. At some point, the system may need to terminate your app to free up memory for the current foreground app—causing any active or pending connections to be lost, for instance. As of iOS 7, Core Bluetooth supports saving state information for central and peripheral manager objects and restoring that state at app launch time. You can use this feature to support long-term actions involving Bluetooth devices.

Foreground-Only Apps

As with most iOS apps, unless you request permission to perform specific background tasks, your app transitions to the suspended state shortly after entering the background state. While in the suspended state, your app is unable to perform Bluetooth-related tasks, nor is it aware of any Bluetooth-related events until it resumes to the foreground.

On the central side, foreground-only apps—apps that have not declared to support either of the Core Bluetooth background execution modes—cannot scan for and discover advertising peripherals while in the background or while suspended. On the peripheral side, advertising is disabled, and any central trying to access a dynamic characteristic value of one of the app's published services receives an error.

Depending on the use case, this default behavior can affect your app in several ways. As an example, imagine that you are interacting with the data on a peripheral that you're currently connected to. Now imagine that your app moves to the suspended state (because, for example, the user switches to another app). If the connection to the peripheral is lost while your app is suspended, you won't be aware that any disconnection occurred until your app resumes to the foreground.

Take Advantage of Peripheral Connection Options

All Bluetooth-related events that occur while a foreground-only app is in the suspended state are queued by the system and delivered to the app only when it resumes to the foreground. That said, Core Bluetooth provides a way to alert the user when certain central role events occur. The user can then use these alerts to decide whether a particular event warrants bringing the app back to the foreground.

You can take advantage of these alerts by including one of the following peripheral connection options when calling the `connectPeripheral:options:` method of the `CBCentralManager` class to connect to a remote peripheral:

- `CBConnectPeripheralOptionNotifyOnConnectionKey`—Include this key if you want the system to display an alert for a given peripheral if the app is suspended when a successful connection is made.
- `CBConnectPeripheralOptionNotifyOnDisconnectionKey`—Include this key if you want the system to display a disconnection alert for a given peripheral if the app is suspended at the time of the disconnection.
- `CBConnectPeripheralOptionNotifyOnNotificationKey`—Include this key if you want the system to display an alert for all notifications received from a given peripheral if the app is suspended at the time.

For more information about the peripheral connection options, see the `Peripheral Connection Options` constants, detailed in *CBCentralManager Class Reference* .

Core Bluetooth Background Execution Modes

If your app needs to run in background to perform certain Bluetooth-related tasks, it must declare that it supports a Core Bluetooth background execution mode in its Information property list (`Info.plist`) file. When your app declares this, the system wakes it up from a suspended state to allow it to handle Bluetooth-related events. This support is important for apps that interact with Bluetooth low energy devices that deliver data at regular intervals, such as a heart rate monitor.

There are two Core Bluetooth background execution modes that an app may declare—one for apps implementing the central role, and another for apps implementing the peripheral role. If your app implements both roles, it may declare that it supports both background execution modes. The Core Bluetooth background execution modes are declared by adding the `UIBackgroundModes` key to your `Info.plist` file and setting the key's value to an array containing one of the following strings:

- `bluetooth-central`—The app communicates with Bluetooth low energy peripherals using the Core Bluetooth framework.
- `bluetooth-peripheral`—The app shares data using the Core Bluetooth framework.

> **Note:** The property list editor in Xcode by default displays human-readable strings for many keys instead of the actual key name. To display the actual key names as they appear in the `Info.plist` file, Control-click any of the keys in the editor window and enable the Show Raw Keys/Values item in the contextual window.

For information about how to configure the contents of your `Info.plist` file, see *Property List Editor Help*.

The bluetooth-central Background Execution Mode

When an app that implements the central role includes the `UIBackgroundModes` key with the `bluetooth-central` value in its `Info.plist` file, the Core Bluetooth framework allows your app to run in the background to perform certain Bluetooth-related tasks. While your app is in the background you can still discover and connect to peripherals, and explore and interact with peripheral data. In addition, the system wakes up your app when any of the `CBCentralManagerDelegate` or `CBPeripheralDelegate` delegate methods are invoked, allowing your app to handle important central role events, such as when a connection is established or torn down, when a peripheral sends updated characteristic values, and when a central manager's state changes.

Although you can perform many Bluetooth-related tasks while your app is in the background, keep in mind that scanning for peripherals while your app is in the background operates differently than when your app is in the foreground. In particular, when your app is scanning for device while in the background:

- The `CBCentralManagerScanOptionAllowDuplicatesKey` scan option key is ignored, and multiple discoveries of an advertising peripheral are coalesced into a single discovery event.
- If all apps that are scanning for peripherals are in the background, the interval at which your central device scans for advertising packets increases. As a result, it may take longer to discover an advertising peripheral.

These changes help minimize radio usage and improve the battery life on your iOS device.

The bluetooth-peripheral Background Execution Mode

To perform certain peripheral role tasks while in the background, you must include the `UIBackgroundModes` key with the `bluetooth-peripheral` value in your app's `Info.plist` file. When this key-value pair is included in the app's `Info.plist` file, the system wakes up your app to process read, write, and subscription events.

In addition to allowing your app to be woken up to handle read, write, and subscription requests from connected centrals, the Core Bluetooth framework allows your app to advertise while in the background state. That said, you should be aware that advertising while your app is in the background operates differently than when your app is in the foreground. In particular, when your app is advertising while in the background:

- The `CBAdvertisementDataLocalNameKey` advertisement key is ignored, and the local name of peripheral is not advertised.
- All service UUIDs contained in the value of the `CBAdvertisementDataServiceUUIDsKey` advertisement key are placed in a special "overflow" area; they can be discovered only by an iOS device that is explicitly scanning for them.
- If all apps that are advertising are in the background, the frequency at which your peripheral device sends advertising packets may decrease.

Use Background Execution Modes Wisely

Although declaring your app to support one or both of the Core Bluetooth background execution modes may be necessary to fulfill a particular use case, you should always perform background processing responsibly. Because performing many Bluetooth-related tasks require the active use of an iOS device's onboard radio—and, in turn, radio usage has an adverse effect on an iOS device's battery life—try to minimize the amount of work you do in the background. Apps woken up for any Bluetooth-related events should process them and return as quickly as possible so that the app can be suspended again.

Any app that declares support for either of the Core Bluetooth background executions modes must follow a few basic guidelines:

- Apps should be session based and provide an interface that allows the user to decide when to start and stop the delivery of Bluetooth-related events.
- Upon being woken up, an app has around 10 seconds to complete a task. Ideally, it should complete the task as fast as possible and allow itself to be suspended again. Apps that spend too much time executing in the background can be throttled back by the system or killed.
- Apps should not use being woken up as an opportunity to perform extraneous tasks that are unrelated to why the app was woken up by the system.

For more-general information about how your app should be behave in the background state, see Being a Responsible Background App in *App Programming Guide for iOS*.

Performing Long-Term Actions in the Background

Some apps may need to use the Core Bluetooth framework to perform long-term actions in the background. As an example, imagine you are developing a home security app for an iOS device that communicates with a door lock (equipped with Bluetooth low energy technology). The app and the lock interact to automatically lock the door when the user leaves home and unlock the door when the user returns—all while the app is in

the background. When the user leaves home, the iOS device may eventually become out of range of the lock, causing the connection to the lock to be lost. At this point, the app can simply call the `connectPeripheral:options:` method of the `CBCentralManager` class, and because connection requests do not time out, the iOS device will reconnect when the user returns home.

Now imagine that the user is away from home for a few days. If the app is terminated by the system while the user is away, the app will not be able to reconnect to the lock when the user returns home, and the user may not be able to unlock the door. For apps like these, it is critical to be able to continue using Core Bluetooth to perform long-term actions, such as monitoring active and pending connections.

State Preservation and Restoration

Because state preservation and restoration is built in to Core Bluetooth, your app can opt in to this feature to ask the system to preserve the state of your app's central and peripheral managers and to continue performing certain Bluetooth-related tasks on their behalf, even when your app is no longer running. When one of these tasks completes, the system relaunches your app into the background and gives your app the opportunity to restore its state and to handle the event appropriately. In the case of the home security app described above, the system would monitor the connection request, and re-relaunch the app to handle the `centralManager:didConnectPeripheral:` delegate callback when the user returned home and the connection request completed.

Core Bluetooth supports state preservation and restoration for apps that implement the central role, peripheral role, or both. When your app implements the central role and adds support for state preservation and restoration, the system saves the state of your central manager object when the system is about to terminate your app to free up memory (if your app has multiple central managers, you can choose which ones you want the system to keep track of). In particular, for a given `CBCentralManager` object, the system keeps track of:

- The services the central manager was scanning for (and any scan options specified when the scan started)
- The peripherals the central manager was trying to connect to or had already connected to
- The characteristics the central manager was subscribed to

Apps that implement the peripheral role can likewise take advantage of state preservation and restoration. For `CBPeripheralManager` objects, the system keeps track of:

- The data the peripheral manager was advertising
- The services and characteristics the peripheral manager published to the device's database
- The centrals that were subscribed to your characteristics' values

When your app is relaunched into the background by the system (because a peripheral your app was scanning for is discovered, for instance), you can reinstantiate your app's central and peripheral managers and restore their state. The following section describes in detail how to take advantage of state preservation and restoration in your app.

Adding Support for State Preservation and Restoration

State preservation and restoration in Core Bluetooth is an opt-in feature and requires help from your app to work. You can add support for this feature in your app by following this process:

1. (Required) Opt in to state preservation and restoration when you allocate and initialize a central or peripheral manager object. This step is described in Opt In to State Preservation and Restoration.
2. (Required) Reinstantiate any central or peripheral manager objects after your app is relaunched by the system. This step is described in Reinstantiate Your Central and Peripheral Managers (page 41).
3. (Required) Implement the appropriate restoration delegate method. This step is described in Implement the Appropriate Restoration Delegate Method (page 42).
4. (Optional) Update your central and peripheral managers' initialization process. This step is described in Update Your Initialization Process (page 43).

Opt In to State Preservation and Restoration

To opt in to the state preservation and restoration feature, simply provide a unique restoration identifier when you allocate and initialize a central or peripheral manager. A *restoration identifier* is a string that identifies the central or peripheral manager to Core Bluetooth and to your app. The value of the string is significant only to your code, but the presence of this string tells Core Bluetooth that it needs to preserve the state of the tagged object. Core Bluetooth preserves the state of only those objects that have a restoration identifier.

For example, to opt in to state preservation and restoration in an app that uses only one instance of a `CBCentralManager` object to implement the central role, specify the `CBCentralManagerOptionRestoreIdentifierKey` initialization option and provide a restoration identifier for the central manager when you allocate and initialize it.

```
myCentralManager =
    [[CBCentralManager alloc] initWithDelegate:self queue:nil
     options:@{ CBCentralManagerOptionRestoreIdentifierKey:
     @"myCentralManagerIdentifier" }];
```

Although the above example does not demonstrate this, you opt in to state preservation and restoration in an app that uses peripheral manager objects in an analogous way: Specify the `CBPeripheralManagerOptionRestoreIdentifierKey` initialization option, and provide a restoration identifier when you allocate and initialize each peripheral manager object.

> **Note:** Because apps can have multiple instances of `CBCentralManager` and `CBPeripheralManager` objects, be sure each restoration identifier is unique, so that the system can properly distinguish one central (or peripheral) manager object from another.

Reinstantiate Your Central and Peripheral Managers

When your app is relaunched into the background by the system, the first thing you need to do is reinstantiate the appropriate central and peripheral managers with the same restoration identifiers as they had when they were first created. If your app uses only one central or peripheral manager, and that manager exists for the lifetime of your app, there is nothing more you need to do for this step.

If your app uses more than one central or peripheral manager or if it uses a manager that isn't around for the lifetime of your app, your app needs to know which managers to reinstantiate when it is relaunched by the system. You can access a list of all the restoration identifiers for the manager objects the system was preserving for your app when it was terminated, by using the appropriate launch option keys (`UIApplicationLaunchOptionsBluetoothCentralsKey` or `UIApplicationLaunchOptionsBluetoothPeripheralsKey`) when implementing your app delegate's `application:didFinishLaunchingWithOptions:` method.

For example, when your app is relaunched by system, you can retrieve all the restoration identifiers for the central manager objects the system was preserving for your app, like this:

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

    NSArray *centralManagerIdentifiers =
        launchOptions[UIApplicationLaunchOptionsBluetoothCentralsKey];
    ...
```

After you have the list of restoration identifiers, simply loop through it and reinstantiate the appropriate central manager objects.

> **Note:** When your app is relaunched, the system provides restoration identifiers only for central and peripheral managers for which it was performing some Bluetooth-related task (while the app was no longer running). These launch option keys are described in more detail in *UIApplicationDelegate Protocol Reference*.

Implement the Appropriate Restoration Delegate Method

After you have reinstantiated the appropriate central and peripheral managers in your app, restore them by synchronizing their state with the state of the Bluetooth system. To bring your app up to speed with what the system has been doing on its behalf (while it was not running), you must implement the appropriate restoration delegate method. For central managers, implement the `centralManager:willRestoreState:` delegate method; for peripheral managers, implement the `peripheralManager:willRestoreState:` delegate method.

> **Important:** For apps that opt in to the state preservation and restoration feature of Core Bluetooth, these are the *first* methods (`centralManager:willRestoreState:` and `peripheralManager:willRestoreState:`) invoked when your app is relaunched into the background to complete some Bluetooth-related task. For apps that don't opt in to state preservation (or if there is nothing to restore upon launch), the `centralManagerDidUpdateState:` and `peripheralManagerDidUpdateState:` delegate methods are invoked first instead.

In both of the above delegate methods, the last parameter is a dictionary that contains information about the managers that were preserved at the time the app was terminated. For a list of the available dictionary keys, see the `Central Manager State Restoration Options` constants in *CBCentralManagerDelegate Protocol Reference* and the `Peripheral_Manager_State_Restoration_Options` constants in *CBPeripheralManagerDelegate Protocol Reference*.

To restore the state of a `CBCentralManager` object, use the keys to the dictionary that is provided in the `centralManager:willRestoreState:` delegate method. As an example, if your central manager object had any active or pending connections at the time your app was terminated, the system continued to monitor them on your app's behalf. As the following shows, you can use the `CBCentralManagerRestoredStatePeripheralsKey` dictionary key to get of a list of all the peripherals (represented by `CBPeripheral` objects) the central manager was connected to or was trying to connect to:

```
- (void)centralManager:(CBCentralManager *)central
      willRestoreState:(NSDictionary *)state {

    NSArray *peripherals =
```

```
        state[CBCentralManagerRestoredStatePeripheralsKey];
    ...
```

What you do with the list of restored peripherals in the above example depends on the use case. For instance, if your app keeps a list of the peripherals the central manager discovers, you may want to add the restored peripherals to that list to keep references to them. As described in Connecting to a Peripheral Device After You've Discovered It (page 19), be sure to set a peripheral's delegate to ensure that it receives the appropriate callbacks.

You can restore the state of a `CBPeripheralManager` object in a similar way by using the keys to the dictionary that is provided in the `peripheralManager:willRestoreState:` delegate method.

Update Your Initialization Process

After you have implemented the previous three required steps, you may want to take a look at updating your central and peripheral managers' initialization process. Although this is an optional step, it can be important in ensuring that things run smoothly in your app. As an example, your app may have been terminated while it was in the middle of exploring the data of a connected peripheral. When your app is restored with this peripheral, it won't know how far it made it the discovery process at the time it was terminated. You'll want to make sure you're starting from where you left off in the discovery process.

For example, when initializing your app in the `centralManagerDidUpdateState:` delegate method, you can find out if you successfully discovered a particular service of a restored peripheral (before your app was terminated), like this:

```
    NSUInteger serviceUUIDIndex =
        [peripheral.services indexOfObjectPassingTest:^BOOL(CBService *obj,
        NSUInteger index, BOOL *stop) {
            return [obj.UUID isEqual:myServiceUUIDString];
        }];

    if (serviceUUIDIndex == NSNotFound) {
        [peripheral discoverServices:@[myServiceUUIDString]];
        ...
```

As the above example shows, if the system terminated your app before it finished discovering the service, begin the exploring the restored peripheral's data at that point by calling the `discoverServices:`. If your app discovered the service successfully, you can then check to see whether the appropriate characteristics were discovered (and whether you already subscribed to them). By updating your initialization process in this manner, you'll ensure that you're calling the right methods at the right time.

Best Practices for Interacting with a Remote Peripheral Device

SwiftObjective-C

The Core Bluetooth framework makes many of the central-side transactions transparent to your app. That is, your app has control over, and is responsible for, implementing most aspects of the central role, such as device discovery and connectivity, and exploring and interacting with a remote peripheral's data. This chapter provides guidelines and best practices for harnessing this level of control in a responsible way, especially when developing your app for an iOS device.

Be Mindful of Radio Usage and Power Consumption

When developing an app that interacts with Bluetooth low energy devices, remember that Bluetooth low energy communication shares your device's radio to transmit signals over the air. Since other forms of wireless communication may need to use your device's radio—for instance, Wi-Fi, classic Bluetooth, and even other apps using Bluetooth low energy—develop your app to minimize how much it uses the radio.

Minimizing radio usage is especially important when developing an app for an iOS device, because radio usage has an adverse effect on an iOS device's battery life. The following guidelines will help you be a good citizen of your device's radio. As a result, your app will perform better and your device's battery will last longer.

Scan for Devices Only When You Need To

When you call the `scanForPeripheralsWithServices:options:` method of the `CBCentralManager` class to discover remote peripheral's that are advertising services, your central device uses its radio to listen for advertising devices until you explicitly tell it to stop.

Unless you need to discover more devices, stop scanning for other devices after you have found one you want to connect to. Use the `stopScan` method of the `CBCentralManager` class to stop scanning for other devices, as shown in Connecting to a Peripheral Device After You've Discovered It (page 19).

Specify the CBCentralManagerScanOptionAllowDuplicatesKey Option Only When Necessary

Remote peripheral devices may send out multiple advertising packets per second to announce their presence to listening centrals. When you are scanning for devices using the `scanForPeripheralsWithServices:options:` method, the default behavior of the method is to coalesce

multiple discoveries of an advertising peripheral into a single discovery event—that is, the central manager calls the `centralManager:didDiscoverPeripheral:advertisementData:RSSI:` method of its delegate object for each new peripheral it discovers, regardless of how many advertising packets it receives. The central manager also calls this delegate method when the advertisement data of an already-discovered peripheral changes.

If you want to change the default behavior, you can specify the `CBCentralManagerScanOptionAllowDuplicatesKey` constant as a scan option when calling the `scanForPeripheralsWithServices:options:` method. When you do, a discovery event is generated each time the central receives an advertising packet from the peripheral. Turning off the default behavior can be useful for certain use cases, such as initiating a connection to a peripheral based on the peripheral's proximity (using the peripheral received signal strength indicator (RSSI) value). That said, keep in mind that specifying this scan option may have an adverse effect on battery life and app performance. Therefore, specify this scan option only when it is necessary to fulfill a particular use case.

Explore a Peripheral's Data Wisely

A peripheral device may have many more services and characteristics than you may be interested in when you are developing an app to fulfill a specific use case. Discovering all of a peripheral's services and associated characteristics can negatively affect battery life and your app's performance. Therefore, you should look for and discover only the services and associated characteristics your app needs.

For example, imagine that you are connected to a peripheral device that has many services available, but your app needs access to only two of them. You can look for and discover these two services only, by passing in an array of their service UUIDs (represented by `CBUUID` objects) to the `discoverServices:` method of the `CBPeripheral` class, like this:

```
[peripheral discoverServices:@[firstServiceUUID, secondServiceUUID]];
```

After you have discovered the two services you are interested in, you can similarly look for and discover only the characteristics of these services that you are interested in. Again, simply pass in an array of the UUIDs that identify the characteristics you want to discover (for each service) to the `discoverCharacteristics:forService:` method of the `CBPeripheral` class.

Subscribe to Characteristic Values That Change Often

As described in Retrieving the Value of a Characteristic (page 21), there are two ways you can retrieve a characteristic's value:

- You can explicitly poll for a characteristic's value by calling the `readValueForCharacteristic:` method each time you need the value.

- You can subscribe to the characteristic's value by calling the `setNotifyValue:forCharacteristic:` method once to receive a notification from the peripheral when the value changes.

It is best practice to subscribe to a characteristic's value when possible, especially for characteristic values that change often. For an example of how to subscribe to a characteristic's value, see Subscribing to a Characteristic's Value (page 22).

Disconnect from a Device When You Have All the Data You Need

You can help reduce your app's radio usage by disconnecting from a peripheral device when a connection is no longer needed. You should disconnect from a peripheral device in both of the following situations:

- All characteristic values that you've subscribed to have stopped sending notifications. (You can determine whether a characteristic's value is notifying by accessing the characteristic's `isNotifying` property.)
- You have all of the data you need from the peripheral device.

In both cases, cancel any subscriptions you may have and then disconnect from the peripheral. You can cancel any subscription to a characteristic's value by calling the `setNotifyValue:forCharacteristic:` method, setting the first parameter to `NO`. You can cancel a connection to a peripheral device by calling the `cancelPeripheralConnection:` method of the `CBCentralManager` class, like this:

```
[myCentralManager cancelPeripheralConnection:peripheral];
```

Note: The `cancelPeripheralConnection:` method is nonblocking, and any `CBPeripheral` class commands that are still pending to the peripheral you're trying to disconnect may or may not finish executing. Because other apps may still have a connection to the peripheral, canceling a local connection does not guarantee that the underlying physical link is immediately disconnected. From your app's perspective, however, the peripheral is considered disconnected, and the central manager object calls the `centralManager:didDisconnectPeripheral:error:` method of its delegate object.

Reconnecting to Peripherals

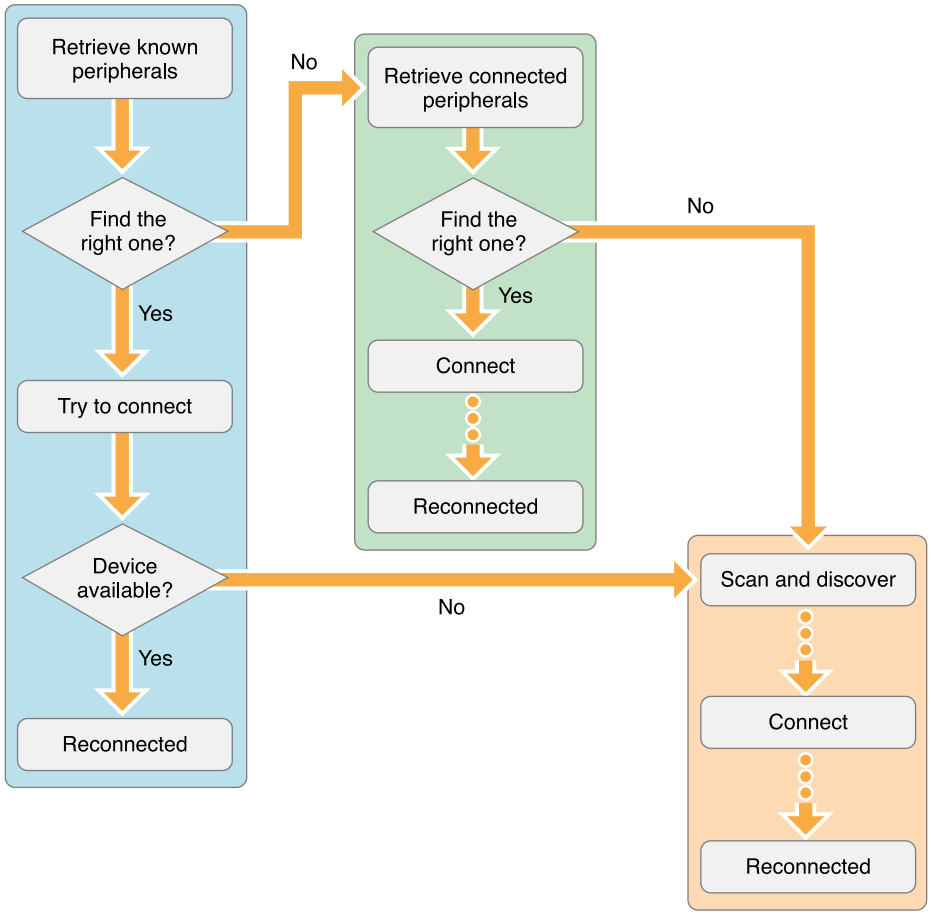
Using the Core Bluetooth framework, there are three ways you can reconnect to a peripheral. You can:

- Retrieve a list of known peripherals—peripherals that you've discovered or connected to in the past—using the `retrievePeripheralsWithIdentifiers:` method. If the peripheral you're looking for is in the list, try to connect to it. This reconnection option is described in Retrieving a List of Known Peripherals (page 49).

- Retrieve a list of peripheral devices that are currently connected to the system using the `retrieveConnectedPeripheralsWithServices:` method. If the peripheral you're looking for is in the list, connect it locally to your app. This reconnection option is described in Retrieving a List of Connected Peripherals (page 50).
- Scan for and discover a peripheral using the `scanForPeripheralsWithServices:options:` method. If you find it, connect to it. These steps are described in Discovering Peripheral Devices That Are Advertising (page 18) and Connecting to a Peripheral Device After You've Discovered It (page 19).

Depending on the use case, you may not want to have to scan for and discover the same peripheral every time you want to reconnect to it. Instead, you may want to try to reconnect using the other options first. As Figure 5-1 shows, one possible reconnection workflow may be to try each of these options in the order in which they're listed above.

Figure 5-1 A sample reconnection workflow



Note: The number of reconnection options you decide to try, and the order in which you do so, may vary by the use case your app is trying to fulfill. For example, you may decide not to use the first connection option at all, or you may decide to try the first two options in parallel.

Retrieving a List of Known Peripherals

The first time you discover a peripheral, the system generates an identifier (a UUID, represented by an `NSUUID` object) to identify the peripheral. You can then store this identifier (using, for instance, the resources of the `NSUserDefaults` class), and later use it to try to reconnect to the peripheral using the `retrievePeripheralsWithIdentifiers:` method of the `CBCentralManager` class. The following describes one way to use this method to reconnect to a peripheral you've previously connected to.

When your app launches, call the `retrievePeripheralsWithIdentifiers:` method, passing in an array containing the identifiers of the peripherals you've previously discovered and connected to (and whose identifiers you have saved), like this:

```
knownPeripherals =
    [myCentralManager retrievePeripheralsWithIdentifiers:savedIdentifiers];
```

The central manager tries to match the identifiers you provided to the identifiers of previously discovered peripherals and returns the results as an array of `CBPeripheral` objects. If no matches are found, the array is empty and you should try one of the other two reconnection options. If the array is not empty, let the user select (in the UI) which peripheral to try to reconnect to.

When the user selects a peripheral, try to connect to it by calling the `connectPeripheral:options:` method of the `CBCentralManager` class. If the peripheral device is still available to be connected to, the central manager calls the `centralManager:didConnectPeripheral:` method of its delegate object and the peripheral device is successfully reconnected.

Note: A peripheral device may not be available to be connected to for a few reasons. For instance, the device may not be in the vicinity of the central. In addition, some Bluetooth low energy devices use a random device address that changes periodically. Therefore, even if the device is nearby, the address of the device may have changed since the last time it was discovered by the system, in which case the `CBPeripheral` object you are trying to connect to doesn't correspond to the actual peripheral device. If you cannot reconnect to the peripheral because its address has changed, you must rediscover it using the `scanForPeripheralsWithServices:options:` method.

For more information about random device addresses, see the Bluetooth 4.0 specification, Volume 3, Part C, Section 10.8 and Bluetooth Accessory Design Guidelines for Apple Products.

Retrieving a List of Connected Peripherals

Another way to reconnect to a peripheral is by checking to see whether the peripheral you're looking for is already connected to the system (for instance, by another app). You can do so by calling the `retrieveConnectedPeripheralsWithServices:` method of the `CBCentralManager` class, which returns an array of `CBPeripheral` objects representing peripheral devices that are currently connected to the system.

Because there may be more than one peripheral currently connected to the system, you can pass in an array of `CBUUID` objects (these object represent service UUIDs) to retrieve only peripherals that are currently connected to the system *and* contain any services that are identified by the UUIDs you specified. If there are no peripheral devices currently connected to the system, the array is empty and you should try one of the other two reconnection options. If the array is not empty, let the user select (in the UI) which one to try to reconnect to.

Assuming that the user finds and selects the desired peripheral, connect it locally to your app by calling the `connectPeripheral:options:` method of the `CBCentralManager` class. (Even though the device is already connected to the system, you must still connect it locally to your app to begin exploring and interacting with it.) When the local connection is established, the central manager calls the `centralManager:didConnectPeripheral:` method of its delegate object, and the peripheral device is successfully reconnected.

Best Practices for Setting Up Your Local Device as a Peripheral

As with many central-side transactions, the Core Bluetooth framework give you control over implementing most aspects of the peripheral role. This chapter provides guidelines and best practices for harnessing this level of control in a responsible way.

Advertising Considerations

Advertising peripheral data is an important part of setting up your local device to implement the peripheral role. The following sections assist you in doing so in an appropriate way.

Respect the Limits of Advertising Data

You advertise your peripheral's data by passing in a dictionary of advertising data to the `startAdvertising:` method of the `CBPeripheralManager` class, as described in Advertising Your Services (page 29). When you create an advertising dictionary, keep in mind that there are limits to what, as well as how much, you can advertise.

Although advertising packets in general can hold a variety of information about the peripheral device, you may advertise only your device's local name and the UUIDs of any services you want to advertise. That is, when you create your advertising dictionary, you may specify only the following two keys: `CBAdvertisementDataLocalNameKey` and `CBAdvertisementDataServiceUUIDsKey`. You receive an error if you specify any other keys.

There are also limits as to how much space you can use when advertising data. When your app is in the foreground, it can use up to 28 bytes of space in the initial advertisement data for any combination of the two supported advertising data keys. If this space is used up, there are an additional 10 bytes of space in the scan response that can be used only for the local name. Any service UUIDs that do not fit in the allotted space are added to a special "overflow" area; they can be discovered only by an iOS device that is explicitly scanning for them. While your app is in the background, the local name is not advertised and all service UUIDs are place in the overflow area.

> **Note:** These sizes do not include the 2 bytes of header information that are required for each new data type. The exact format of advertising and response data is defined in the Bluetooth 4.0 specification, Volume 3, Part C, Section 11.

To help you stay within these space constraints, limit the service UUIDs you advertise to those that identify your primary services.

Advertise Data Only When You Need To

Since advertising peripheral data uses your local device's radio (and thus your device's battery), advertise only when you want other devices to connect to you. Once connected, these devices can explore and interact the peripheral's data directly, without the need for any advertising packets. Therefore, to minimize radio usage, increase app performance, and preserve your device's battery, stop advertising when it is no longer necessary to facilitate any intended Bluetooth low energy transaction. To stop advertising on your local peripheral, simply call the `stopAdvertising` method of the `CBPeripheralManager` class, like this:

```
[myPeripheralManager stopAdvertising];
```

Let the User Decide When to Advertise

Knowing when to advertise is often something only the user can know. For example, it doesn't make sense to have your app advertise services on your device when you know there aren't any other Bluetooth low energy devices nearby. Since your app is often unaware of what other devices are nearby, provide in your app's user interface (UI) a way for the user to decide when to advertise.

Configuring Your Characteristics

When you create a mutable characteristic, you set its properties, value, and permissions. These settings determine how connected centrals access and interact with the characteristic's value. Although you may decide to configure the properties and permissions of your characteristics differently based on the needs of your app, the following sections provide some guidance when you need to perform the following two tasks:

- Allow connected centrals to subscribe to your characteristics
- Protect sensitive characteristic values from being accessed by unpaired centrals

Configure Your Characteristics to Support Notifications

As described in Subscribe to Characteristic Values That Change Often (page 46), it is recommended that centrals subscribe to characteristic values (of a remote peripheral's service) that change often. When possible, encourage this practice by allowing connected centrals to subscribe to your characteristics' values.

When you create a mutable characteristic, configure it to support subscriptions by setting the characteristic's properties with the `CBCharacteristicPropertyNotify` constant, like this:

```
myCharacteristic = [[CBMutableCharacteristic alloc]
    initWithType:myCharacteristicUUID
    properties:CBCharacteristicPropertyRead | CBCharacteristicPropertyNotify
    value:nil permissions:CBAttributePermissionsReadable];
```

In this example, the characteristic's value is readable, and it can be subscribed to by a connected central.

Require a Paired Connection to Access Sensitive Data

Depending on the use case, you may want to vend a service that has one or more characteristic whose value needs to be secure. For example, imagine that you want to vend a social media profile service. This service may have characteristics whose values represent a member's profile information, such as first name, last name, and email address. More than likely, you want to allow only trusted devices to retrieve a member's email address.

You can ensure that only trusted devices have access to sensitive characteristic values by setting the appropriate characteristic properties and permissions. To continue the example above, to allow only trusted devices to retrieve a member's email address, set the appropriate characteristic's properties and permissions, like this:

```
emailCharacteristic = [[CBMutableCharacteristic alloc]
    initWithType:emailCharacteristicUUID
    properties:CBCharacteristicPropertyRead
    | CBCharacteristicPropertyNotifyEncryptionRequired
    value:nil permissions:CBAttributePermissionsReadEncryptionRequired];
```

In this example, the characteristic is configured to allow only trusted devices to read or subscribe to its value. When a connected, remote central tries to read or subscribe to this characteristic's value, Core Bluetooth tries to pair your local peripheral with the central to create a secure connection.

For example, if the central and the peripheral are iOS devices, both devices receive an alert indicating that the other device would like to pair. The alert on the central device contains a code that you must enter into a text field on the peripheral device's alert to complete the pairing process.

After the pairing process is complete, the peripheral considers the paired central a trusted device and allows the central access to its encrypted characteristic values.

Document Revision History

This table describes the changes to *Core Bluetooth Programming Guide*.

Date	Notes
2013-09-18	Updated for iOS 7 and OS X v10.9 with information about performing long-term actions in the background and new methods for retrieving peripherals.
2013-08-08	New document that describes how to use the Core Bluetooth framework to develop apps that interact with Bluetooth low energy technology.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010