Working With Bluetooth Devices

Preliminary 2004-06-29

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

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to Working With Bluetooth Devices

Note: This is a preliminary document for an API in development. Although this document has been reviewed for technical accuracy, it is not final. Apple Computer is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API. For information about updates to this and other developer documentation, view the New & Updated sidebars in subsequent seeds of the Reference Library.

This document describes how Bluetooth works and summarizes the Bluetooth specification in order to provide a foundation for understanding Apple's Bluetooth support. It also provides task information, accompanied by several code examples, that illustrates how to develop applications that access Bluetooth-enabled devices.

What Is Bluetooth?

Bluetooth is an open specification that enables low-bandwidth, short-range wireless connections between computers and peripherals, such as mice, cell phones, and personal data assistants (PDAs). The appeal of the Bluetooth model lies in its convenience for wirelessly transferring information and small data files between devices.

Bluetooth is not a networking solution, so it is not a competitor of AirPort, Apple's wireless networking technology. Nor is it a replacement for the cables needed by high-bandwidth peripherals, such as FireWire. Rather, Bluetooth offers a replacement for IrDA (Infrared Data Association) technology, because it is not constrained by IrDA's shorter range and line-of-sight requirements.

Who Should Read This Document?

Because this document comprises conceptual and task information, its audience is broad. This document sets the stage with an overview of Bluetooth technology. Then, it describes how Apple implements the Bluetooth specification and how to access Bluetooth-enabled devices on Mac OS X. If you're unfamiliar with Bluetooth technology in general, you can read this document for a high-level summary. If you're primarily interested in learning about how Apple implements the Bluetooth specification,

you'll find a thorough description in this document. Finally, if you're developing applications that communicate with or control Bluetooth-enabled devices, you should read this document to discover your options.

Apple provides application-level access to its Bluetooth API in both C and Objective-C, so knowledge of one or the other is important for understanding the code samples. Additionally, the Mac OS X Bluetooth model is at heart an object-oriented one, so familiarity with object-oriented principles is helpful.

Organization of This Document

This book is divided into three chapters:

- "Bluetooth Technology Basics" (page 9) describes how Bluetooth works and outlines the Bluetooth protocol stack as defined by the Bluetooth Special Interest Group. If you're already familiar with Bluetooth technology and the architecture of the protocol stack, you may choose to skip ahead to the next chapter.
- "Bluetooth on Mac OS X" (page 19) describes the Mac OS X Bluetooth implementation and outlines the services available to you. This chapter provides a foundation for the specific code samples in the last chapter.
- "Developing Bluetooth Applications" (page 35) describes several common tasks that most Bluetooth applications perform. In addition, it presents a sample application that illustrates how to bring many of these tasks together in a working application. This chapter does not attempt to define the Mac OS X Bluetooth API. For complete API reference, see `/Developer/Documentation/DeviceDriversIOKit/DeviceDriversIOKit.html`.

See Also

Apple provides comprehensive API reference documentation for its Bluetooth support. On the web, this reference is available at Device Drivers Bluetooth Reference Library.

In addition, Apple provides several sample applications that show how to make various Bluetooth connections. These samples are included in the Bluetooth SDK, available on the web at http://developer.apple.com/hardware/bluetooth/.

When you install the Developer package, you get developer documentation as well as tools and example code. The Bluetooth API reference documentation is available in the Device Drivers section of `/Developer/ADC Reference Library/documentation/index.html`. The sample code is in `/Developer/Examples/Bluetooth`. Bluetooth-specific utility applications are in `/Developer/Applications/Utilities/Bluetooth`.

There are many books that describe Bluetooth technology and the Bluetooth specification. A popular one is *Bluetooth: Connect Without Cables* by Jennifer Bray and Charles F. Sturman.

To view the Bluetooth specification itself, see http://www.bluetooth.com. This site also provides information about the Bluetooth Special Interest Group and the product-qualification program.

Bluetooth Technology Basics

This chapter provides an overview of Bluetooth technology, including a summary of the Bluetooth specification. You should read this chapter if you're new to Bluetooth technology or if you need a brief overview of the Bluetooth specification.

Bluetooth Overview

Bluetooth is a cable-replacement technology designed to wirelessly connect peripherals, such as mice and mobile phones, to your desktop or laptop computer and to each other. An inexpensive, low-power, short-range radio-based technology, Bluetooth is not a wireless networking solution, such as AirPort. Rather, it is an alternative to the IrDA (Infrared Data Association) standard. Although the IrDA standard, too, supports wireless communication between peripherals and computers, it has two limiting requirements. First, IrDA devices must be very close, no more than about 1 meter apart. Second, the communicating devices must have a direct line of sight to each other.

Because it relies on radio waves, however, Bluetooth communication overcomes these strict requirements:

- Bluetooth devices can communicate at ranges of up to 10 meters.
- Bluetooth devices do not need to be in direct sight of each other.

This makes Bluetooth communication much more flexible and robust. It's also important to note that because Bluetooth excels at low-bandwidth data transfer, it is not intended as a replacement for high-bandwidth cabled peripherals. For high-bandwidth devices, such as external hard drives or video cameras, cables are still the best option.

Apple's Bluetooth support is integrated into Mac OS X, version 10.2 and later, and is based on the Bluetooth Special Interest Group (SIG) specification (discussed in "Bluetooth Architecture" (page 12)). Apple also provides some high-level bridges between Mac OS X functionality and the Bluetooth protocol stack. This means that many Bluetooth devices work transparently with computers running Mac OS X version 10.2 and later. The Mac OS X HID Manager, for example, handles a Bluetooth mouse just as it does a cabled mouse. In many cases, such high-level bridges allow your application to handle Bluetooth devices without including any Bluetooth-specific code.

Other applications may need to access Bluetooth-specific attributes and messages. For them, Apple provides a comprehensive API that allow you to take advantage of Bluetooth's unique features. Be sure to read "Bluetooth on Mac OS X" (page 19) for a description of the Bluetooth API available in Mac OS X version 10.2 and later. For concrete examples showing how to use Apple's Bluetooth API, see "Developing Bluetooth Applications" (page 35).

What Bluetooth Does Best

The characteristics of Bluetooth technology—low cost, low power, and radio based— encouraged the concept of a personal area network (PAN). A PAN envelops the user in a small, mobile bubble of connectivity that is effortlessly available at any time. Bluetooth's freedom from cables and potential ubiquity make it ideal for carrying your personal network around with you.

With a PAN, the possibilities are limitless:

- Imagine being able to connect to the Internet on a dial-up connection you access through your mobile phone. Surfing the Internet then becomes possible anywhere your mobile phone can connect to your internet service provider.
- Perhaps you prefer to use a traditional mouse with your laptop. Choose a Bluetooth-enabled mouse and you won't have to keep track of a mouse cable.
- If you have a Bluetooth-enabled mobile phone that stores your business information in the Vcard format, you can easily share this information with your colleagues. Swap your Vcard with theirs, by wirelessly connecting to their Bluetooth-enabled mobile phones.

Future Directions for Bluetooth

Industry analysts predict the growing popularity and availability of Bluetooth-enabled devices. This in turn raises consumer expectations for mobile PANs and provides many opportunities for vendors to create new products. At the end of 2003, the Bluetooth SIG released the second version (version 1.2) of the Bluetooth specification. This successor to version 1.1 provides a number of improvements, including:

- Enhanced quality of service (QOS). This guarantees that your human-interface (and other QOS) devices will get the time to transfer data when they need it.
- A more adaptive frequency-hopping algorithm. The new algorithm increases communication reliability and decreases interference from other wireless emitters operating the same frequency range.

Apple's ongoing support for Bluetooth communication is evidenced by frequent Bluetooth software updates and up-to-date SDKs. Using the software frameworks and built-in support Apple provides, you can bring your Bluetooth applications to Mac OS X with ease. Apple is committed to helping you find ways to provide your customers with the wireless connectivity they need.

How Bluetooth Works

This section seeks to give you an overview of the technology and specification that will provide context for the Bluetooth implementation on Mac OS X. If you're already familiar with the Bluetooth specification and how Bluetooth devices work, you might choose to skip ahead to "Bluetooth on Mac OS X" (page 19).

Bluetooth devices operate at 2.4 GHz in the license-free, globally available ISM (Industrial, Scientific, and Medical) radio band. The advantage of operating in this band is worldwide availability and compatibility. A potential disadvantage is that Bluetooth devices must share this band with many other RF emitters. These include automobile security systems, other wireless communications standards (such as 802.11), and ordinary noise sources (such as microwave ovens).

To overcome this challenge, Bluetooth employs a fast frequency-hopping scheme and uses shorter packets than other standards in the ISM band. This scheme makes Bluetooth communication more robust and more secure.

Frequency Hopping

Frequency hopping is literally jumping from frequency to frequency within the ISM band. After a Bluetooth device sends or receives a packet, it and the Bluetooth device or devices it is communicating with "hop" to another frequency before the next packet is sent. This scheme has three advantages:

- It allows Bluetooth devices to use the entirety of the available ISM band, while never transmitting from a fixed frequency for more than a very short time This ensures that Bluetooth conforms to the ISM restrictions on transmission quantity per frequency.
- It ensures that any interference will be short-lived. Any packet that doesn't arrive safely at its destination can be resent at the next frequency.
- It provides a base level of security because it's very difficult for an eavesdropping device to predict which frequency the Bluetooth devices will use next.

Of course, the connected devices must agree upon the next frequency to use. The Bluetooth specification ensures this in two ways. First, it defines a master-slave relationship between Bluetooth devices. Second, it specifies an algorithm that uses device-specific information to calculate frequency-hop sequences.

A Bluetooth device operating in master mode can communicate with up to seven slave devices. To each of its slaves, the master Bluetooth device sends its own unique device address (similar to an ethernet address) and the value of its internal clock. This information is used to calculate the frequency-hop sequence. Because the master device and all its slaves use the same algorithm with the same initial input, the connected devices always arrive together at the next frequency.

Power Consumption

As a cable-replacement technology, it's not surprising that Bluetooth devices are usually battery-powered devices, such as wireless mice and mobile phones. To conserve power, most Bluetooth devices operate as low-power, 1 mW radios (Class 3 radio power). This gives Bluetooth devices a range of about 5–10 meters. This range is far enough for comfortable wireless peripheral communication but close enough to avoid drawing too much power from the device's power source.

Security

Security is a challenge faced by every communications standard. Wireless communications present special security challenges. Bluetooth builds security into its model on several different levels, beginning with the security inherent in its frequency-hopping scheme (described in "Frequency Hopping" (page 11)).

At the lowest levels of the protocol stack, Bluetooth uses the publicly available cipher algorithm known as SAFER+ to authenticate a device's identity. The generic-access profile depends on this authentication for its device-pairing process. This process involves creating a special link to create and exchange a link key. Once verified, the link key is used to negotiate an encryption mode the devices will use for their communication.

The topic of Bluetooth security is beyond the scope of this document. For references that contain more information on Bluetooth's encryption and authentication processes, see "See Also" (page 8).

Bluetooth Architecture

Bluetooth is both a hardware-based radio system and a software stack that specifies the linkages between layers. This supports flexibility in implementation across different devices and platforms. It also provides robust guidelines for maximum interoperability and compatibility.

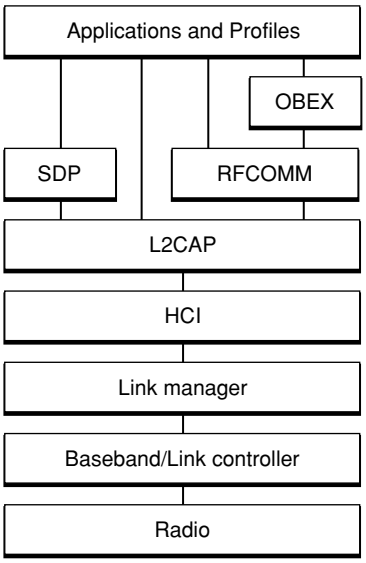
In this section, you'll learn about:

- The **Bluetooth protocol stack**. The protocol stack is the core of the Bluetooth specification that defines how the technology works.
- The **Bluetooth profiles**. The profiles define how to use Bluetooth technology to accomplish specific tasks.

The Bluetooth Protocol Stack

The heart of the Bluetooth specification is the Bluetooth protocol stack. By providing well-defined layers of functionality, the Bluetooth specification ensures interoperability of Bluetooth devices and encourages adoption of Bluetooth technology. As you can see in Figure 1-1 (page 13), these layers range from the low-level radio link to the profiles.

Figure 1-1 The Bluetooth protocol stack



Lower Layers

At the base of the Bluetooth protocol stack is the **radio layer**. The radio module in a Bluetooth device is responsible for the modulation and demodulation of data into RF signals for transmission in the air. The radio layer describes the physical characteristics a Bluetooth device's receiver-transmitter component must have. These include modulation characteristics, radio frequency tolerance, and sensitivity level.

Above the radio layer is the **baseband** and **link controller layer**. The Bluetooth specification doesn't establish a clear distinction between the responsibilities of the baseband and those of the link controller. The best way to think about it is that the baseband portion of the layer is responsible for properly formatting data for transmission to and from the radio layer. In addition, it handles the synchronization of links. The link controller portion of this layer is responsible for carrying out the link manager's commands and establishing and maintaining the link stipulated by the link manager.

The **link manager** itself translates the host controller interface (HCI) commands it receives into baseband-level operations. It is responsible for establishing and configuring links and managing power-change requests, among other tasks.

You've noticed links mentioned numerous times in the preceding paragraphs. The Bluetooth specification defines two types of links between Bluetooth devices:

- **Synchronous, Connection-Oriented (SCO)**, for isochronous and voice communication using, for example, headsets
- **Asynchronous, Connectionless (ACL)**, for data communication, such as the exchange of vCards

Each link type is associated with a specific packet type. A SCO link provides reserved channel bandwidth for communication between a master and a slave, and supports regular, periodic exchange of data with no retransmission of SCO packets.

An ACL link exists between a master and a slave the moment a connection is established. The data packets Bluetooth uses for ACL links all have 142 bits of encoding information in addition to a payload that can be as large as 2712 bits. The extra amount of data encoding heightens transmission security. It also helps to maintain a robust communication link in an environment filled with other devices and common noise.

The **HCI (host controller interface) layer** acts as a boundary between the lower layers of the Bluetooth protocol stack and the upper layers. The Bluetooth specification defines a standard HCI to support Bluetooth systems that are implemented across two separate processors. For example, a Bluetooth system on a computer might use a Bluetooth module's processor to implement the lower layers of the stack (radio, baseband, link controller, and link manager). It might then use its own processor to implement the upper layers (L2CAP, RFCOMM, OBEX, and selected profiles). In this scheme, the lower portion is known as the *Bluetooth module* and the upper portion as the *Bluetooth host*.

Of course, it's not required to partition the Bluetooth stack in this way. Bluetooth headsets, for example, combine the module and host portions of the stack on one processor because they need to be small and self-contained. In such devices, the HCI may not be implemented at all unless device testing is required.

Because the Bluetooth HCI is well defined, you can write drivers that handle different Bluetooth modules from different manufacturers. Apple provides an HCI controller object that supports a USB implementation of the HCI layer.

Upper Layers

Above the HCI layer are the upper layers of the protocol stack. The first of these is the **L2CAP (logical link control and adaptation protocol) layer**. The L2CAP is primarily responsible for:

- Establishing connections across existing ACL links or requesting an ACL link if one does not already exist
- Multiplexing between different higher layer protocols, such as RFCOMM and SDP, to allow many different applications to use a single ACL link
- Repackaging the data packets it receives from the higher layers into the form expected by the lower layers

The L2CAP employs the concept of channels to keep track of where data packets come from and where they should go. You can think of a channel as a logical representation of the data flow between the L2CAP layers in remote devices. Because it plays such a central role in the communication between the upper and lower layers of the Bluetooth protocol stack, the L2CAP layer is a required part of every Bluetooth system.

Above the L2CAP layer, the remaining layers of the Bluetooth protocol stack aren't quite so linearly ordered. However, it makes sense to discuss the service discovery protocol next, because it exists independently of other higher-level protocol layers. In addition, it is common to every Bluetooth device.

The **SDP (service discovery protocol)** defines actions for both servers and clients of Bluetooth services. The specification defines a service as any feature that is usable by another (remote) Bluetooth device. A single Bluetooth device can be both a server and a client of services. An example of this is the Macintosh computer itself. Using the file transfer profile (described in "The Bluetooth Profiles—A Hierarchy of Groups" (page 15)) a Macintosh computer can browse the files on another device and allow other devices to browse its files.

An SDP client communicates with an SDP server using a reserved channel on an L2CAP link to find out what services are available. When the client finds the desired service, it requests a separate connection to use the service. The reserved channel is dedicated to SDP communication so that a device always knows how to connect to the SDP service on any other device. An SDP server maintains its own SDP database, which is a set of service records that describe the services the server offers. Along with information describing how a client can connect to the service, the service record contains the service's **UUID**, or **universally unique identifier**.

Also above the L2CAP layer in Figure 1-1 (page 13) is the **RFCOMM layer**. The RFCOMM protocol emulates the serial cable line settings and status of an RS-232 serial port. RFCOMM connects to the lower layers of the Bluetooth protocol stack through the L2CAP layer.

By providing serial-port emulation, RFCOMM supports legacy serial-port applications. It also supports the OBEX protocol (discussed next) and several of the Bluetooth profiles (discussed in "The Bluetooth Profiles—A Hierarchy of Groups" (page 15)).

OBEX (object exchange) is a transfer protocol that defines data objects and a communication protocol two devices can use to easily exchange those objects. Bluetooth adopted OBEX from the IrDA IrOBEX specification because the lower layers of the IrOBEX protocol are very similar to the lower layers of the Bluetooth protocol stack. In addition, the IrOBEX protocol is already widely accepted and therefore a good choice for the Bluetooth SIG, which strives to promote adoption by using existing technologies.

A Bluetooth device wanting to set up an OBEX communication session with another device is considered to be the client device.

1. The client first sends SDP requests to make sure the other device can act as a server of OBEX services.

2. If the server device can provide OBEX services, it responds with its OBEX service record. This record contains the RFCOMM channel number the client should use to establish an RFCOMM channel.

3. Further communication between the two devices is conveyed in packets, which contain requests and responses, and data. The format of the packet is defined by the OBEX session protocol.

Although OBEX can be supported over TCP/IP, this document does not discuss this option (nor is it described in the Bluetooth specification).

The Bluetooth Profiles—A Hierarchy of Groups

The Bluetooth specification defines a wide range of profiles, describing many different types of tasks, some of which have not yet been implemented by any device or system. By following the profiles's procedures, developers can be sure that the applications they create will work with any device that conforms to the Bluetooth specification.This section focuses on those profiles that Mac OS X supports. For information on other profiles, including those still in development, see the Bluetooth specification.

At a minimum, each profile specification contains information on the following topics:

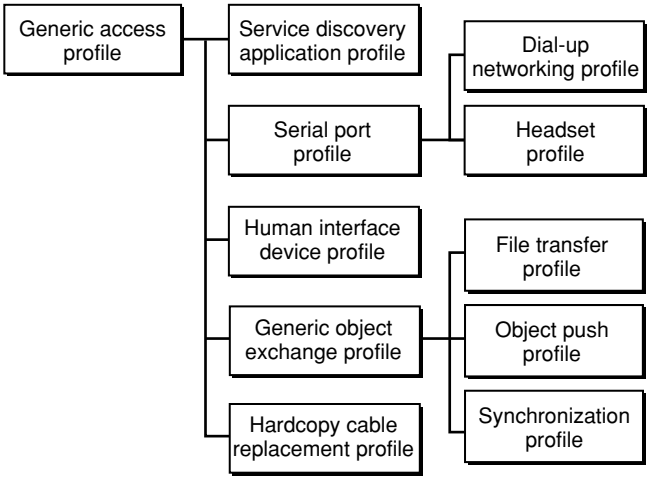
- **Dependencies on other profiles.** Every profile depends on the base profile, called the generic access profile, and some also depend on intermediate profiles.

- **Suggested user interface formats.** Each profile describes how a user should view the profile so that a consistent user experience is maintained.

- **Specific parts of the Bluetooth protocol stack used by the profile.** To perform its task, each profile uses particular options and parameters at each layer of the stack. This may include an outline of the required service record, if appropriate.

The Bluetooth profiles are organized into a hierarchy of groups, with each group depending upon the features provided by its predecessor. Figure 1-2 (page 16) illustrates the dependencies of the Bluetooth profiles.

Figure 1-2 The Bluetooth profiles



The Base Profile

At the base of the profile hierarchy is the generic access profile (GAP), which defines a consistent means to establish a baseband link between Bluetooth devices. In addition to this, the GAP defines:

- Which features must be implemented in all Bluetooth devices
- Generic procedures for discovering and linking to devices
- Basic user-interface terminology

All other profiles are based on the GAP. This allows each profile to take advantage of the features the GAP provides and ensures a high degree of interoperability between applications and devices. It also makes it easier for developers to define new profiles by leveraging existing definitions.

Remaining Profiles

The **service discovery application profile** describes how an application should use the SDP (described in "The Bluetooth Protocol Stack" (page 12)) to discover services on a remote device. As required by the GAP, any Bluetooth device should be able to connect to any other Bluetooth device. Based on this, the service discovery application profile requires that any application be able to find out what services are available on any Bluetooth device it connects to.

The **human interface device (HID) profile** describes how to communicate with a HID class device using a Bluetooth link. It describes how to use the USB HID protocol to discover a HID class device's feature set and how a Bluetooth device can support HID services using the L2CAP layer. For more information about the USB HID protocol, see http://www.usb.org.

As its name suggests, the **serial port profile** defines RS-232 serial-cable emulation for Bluetooth devices. As such, the profile allows legacy applications to use Bluetooth as if it were a serial-port link, without requiring any modification. The serial port profile uses the RFCOMM protocol to provide the serial-port emulation.

The **dial-up networking (DUN) profile** is built on the serial port profile and describes how a data-terminal device, such as a laptop computer, can use a gateway device, such as a mobile phone or a modem, to access a telephone-based network. Like other profiles built on top of the serial port profile, the virtual serial link created by the lower layers of the Bluetooth protocol stack is transparent to applications using the DUN profile. Thus, the modem driver on the data-terminal device is unaware that it is communicating over Bluetooth. The application on the data-terminal device is similarly unaware that it is not connected to the gateway device by a cable.

The **headset profile** describes how a Bluetooth-enabled headset should communicate with a computer or other Bluetooth device (such as a mobile phone). When connected and configured, the headset can act as the remote device's audio input and output interface.

The **hardcopy cable replacement profile** describes how to send rendered data over a Bluetooth link to a device, such as a printer. Although other profiles can be used for printing, the HCRP is specially designed to support hardcopy applications.

The **generic object exchange profile** provides a generic blueprint for other profiles using the OBEX protocol and defines the client and server roles for devices. As with all OBEX transactions, the generic object exchange profile stipulates that the client initiate all transactions. The profile does not, however, describe how applications should define the objects to exchange or exactly how the applications should implement the exchange. These details are left to the profiles that depend on the generic object exchange profile, namely the object push, file transfer, and synchronization profiles.

The **object push profile** defines the roles of push server and push client. These roles are analogous to and must interoperate with the server and client device roles the generic object exchange profile defines. The object push profile focuses on a narrow range of object formats for maximum interoperability. The most common of the acceptable formats is the vCard format. If an application needs to exchange data in other formats, it should use another profile, such as the file transfer profile.

The **file transfer profile** is also dependent on the generic object exchange profile. It provides guidelines for applications that need to exchange objects such as files and folders, instead of the more limited objects supported by the object push profile. The file transfer profile also defines client and server device roles and describes the range of their responsibilities in various scenarios. For example, if a client wishes to browse the available objects on the server, it is required to support the ability to pull from the server a folder-listing object. Likewise, the server is required to respond to this request by providing the folder-listing object.

The **synchronization profile** is another dependent of the generic object exchange profile. It describes how applications can perform data synchronization, such as between a personal data assistant (PDA) and a computer. Not surprisingly, the synchronization profile, too, defines client and server device roles. The synchronization profile focuses on the exchange of personal information management (PIM) data, such as a to-do list, between Bluetooth-enabled devices. A typical usage of this profile would be an application that synchronizes your computer's and your PDA's versions of your PIM

data. The profile also describes how an application can support the automatic synchronization of data—in other words, synchronization that occurs when devices discover each other, rather than at a user's command.

Bluetooth on Mac OS X

Because Apple provides high-level managers and abstractions that transparently perform Bluetooth connection–oriented tasks for many types of applications, you may never need to use the API this chapter describes. There are some exceptions to this, however, such as:

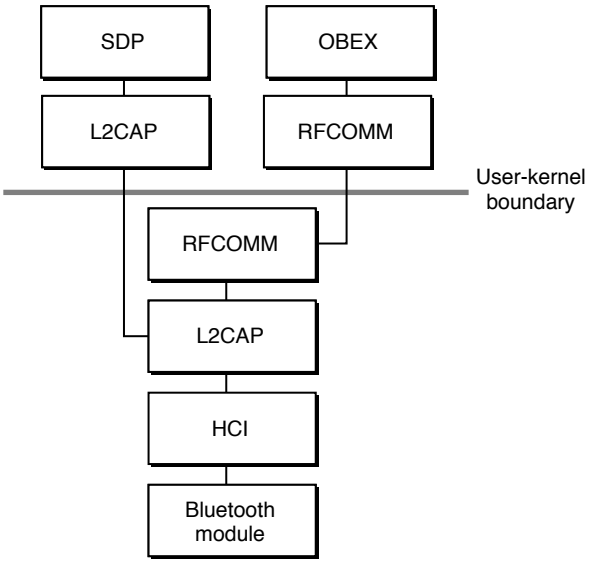
- An application that must display Bluetooth-specific error messages to provide a more informative and responsive user experience
- An application that vends a new service
- An application that implements a new profile

In these applications, you will need to use the API the Bluetooth frameworks provide. To prepare a foundation for the discussion of the API, this chapter first describes the Mac OS X implementation of the Bluetooth protocol stack. Then, it describes the various objects, methods, and functions available in the Bluetooth frameworks. For application-design considerations and outlines of how to use the Bluetooth API to perform specific tasks, see "Developing Bluetooth Applications" (page 35).

The Mac OS X Bluetooth Protocol Stack

The foundation of Mac OS X Bluetooth support is Apple's implementation of the Bluetooth protocol stack. For reference, the Bluetooth protocol stack, as defined by the Bluetooth SIG, is described in "The Bluetooth Protocol Stack" (page 12). Figure 2-1 (page 20) shows the Bluetooth protocol stack that's built into Mac OS X version 10.2 and later.

Figure 2-1 The Mac OS X Bluetooth protocol stack



The Mac OS X implementation of the Bluetooth protocol stack includes both in-kernel and user-level portions. Figure 2-1 (page 20) shows the layers of the stack that exist in the kernel and the corresponding user-level layers an application can access.

The **Bluetooth module** at the bottom of the stack is the hardware component that implements the Bluetooth radio, baseband, and link manager protocols. Neither an application nor even the host has access to this layer of the stack.

As described in "The Bluetooth Protocol Stack" (page 12), the **HCI layer** transmits data and commands from the layers above to the Bluetooth module below. Conversely, the HCI layer receives events from the Bluetooth module and transmits them to the upper layers.

To implement the functions of the HCI layer in the kernel, Apple defines the abstract class IOBluetoothHCIController. IOBluetoothHCIController is the superclass of another in-kernel object, AppleBluetoothUSBHCIController, which provides support for Bluetooth over USB. Therefore, any hardware that supports the USB HCI specification should work with the Bluetooth implementation on Mac OS X version 10.2 and later. It is possible, although certainly not trivial, to subclass the IOBluetoothHCIController object to provide vendor-specific functionality or to support Bluetooth over a transport other than USB. If you need to do this, you should contact Apple's Developer Technical Support (at http://developer.apple.com/technicalsupport) for assistance.

Apple implements the L2CAP and RFCOMM layers in the kernel. Applications can use objects in the user-level L2CAP and RFCOMM layers to access the corresponding in-kernel objects, although many applications will not need to do so directly.

Recall that the L2CAP (logical link control adaptation protocol) provides:

- Multiplexing of data channels
- Segmentation and reassembly of data packets to conform to a device's maximum packet size
- Support for different channel types and channel IDs, such as RFCOMM

The in-kernel **L2CAP layer** provides the transport for the higher-level protocols and profiles. As the primary communication gateway between two Bluetooth-enabled devices, the Mac OS X L2CAP layer implements the ability to register as a client of an L2CAP channel and write data to the channel. Using the L2CAP layer's multiplexing feature, it is possible to send and receive data to and from the RFCOMM layer and the SDP layer at the same time.

Above the in-kernel L2CAP layer in Figure 2-1 (page 20) is the RFCOMM protocol layer. The in-kernel **RFCOMM protocol layer** is a serial-port emulation protocol. Its primary mission is to make a data channel appear as a serial port. It also implements the ability to create and destroy RFCOMM channels and to control the speed of the channel as if it were a physical serial-port cable.

The portion of the stack shown above the user-kernel boundary in Figure 2-1 (page 20) is accessible to applications. The L2CAP and RFCOMM layers in user space are not duplicates of the in-kernel L2CAP and RFCOMM layers. Instead, they represent the user-level APIs an application uses to communicate with the corresponding in-kernel layers.

The **SDP (service discovery protocol) layer** is more of a service than a protocol, but it is part of the built-in Mac OS X Bluetooth protocol stack. It is shown connected to the user-level L2CAP layer because it uses an L2CAP channel to communicate with remote Bluetooth devices to discover their available services. Apple provides an SDP API you can use to discover what services a device supports.

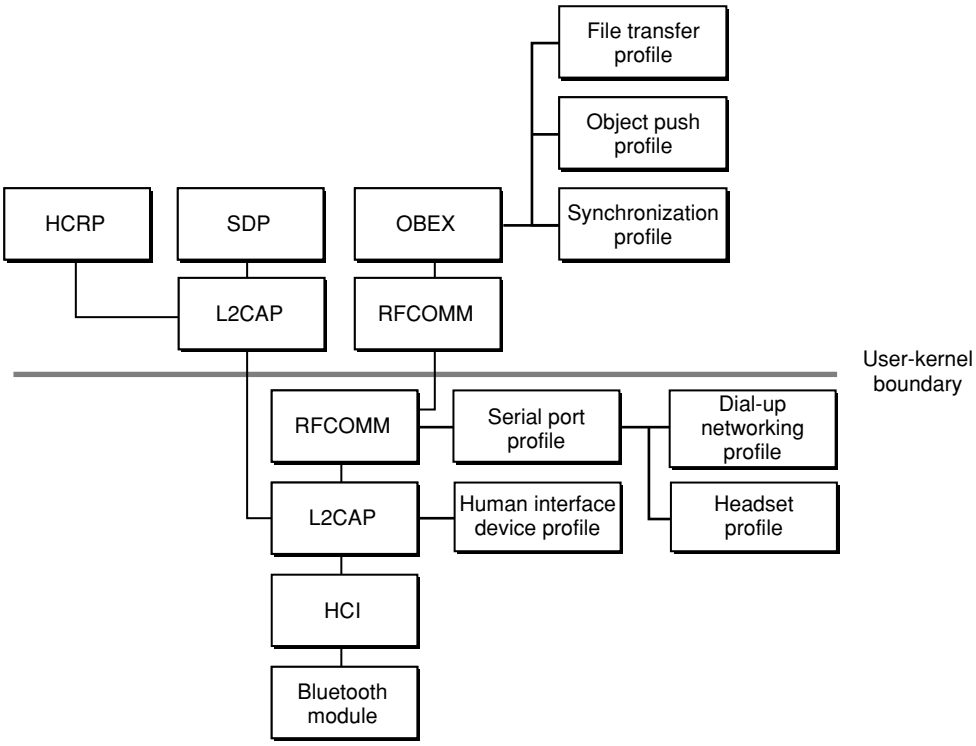
Above the user-level RFCOMM layer is the **OBEX (object exchange) protocol layer**. The OBEX protocol is an HTTP-like protocol that supports the transfer of simple objects, like files, between devices. The OBEX protocol uses an RFCOMM channel for transport because of the similarities between IrDA (which defines the OBEX protocol) and serial-port communication.

In versions of Mac OS X prior to 10.4, the OBEX API provides primitive commands, such as `CONNECT`, `PUT`, and `GET`. Beginning in Mac OS X version 10.4, Apple introduced API that simplifies common OBEX operations found in the object push and file transfer profiles, such as getting and putting file objects on remote devices. This API makes it easy for a developer to create an application that adheres to the requirements of these profiles. Note that if you want to use OBEX for other profiles, however, you probably will still need to rely on the primitive commands provided by the OBEX API. For more information on the new API, see "OBEXFileTransferServices Class" (page 26) and "Objects in OBEX Connections" (page 29).

The Mac OS X Bluetooth Profiles and Applications

In addition to the protocols shown in Figure 2-1 (page 20), Mac OS X implements several Bluetooth profiles. In general, a profile defines a particular usage of the protocols. Figure 2-2 (page 22) shows how each profile is built on top of particular protocols in the Mac OS X Bluetooth protocol stack.

Figure 2-2 Mac OS X Bluetooth profiles



In Mac OS X version 10.2.8 and later, the available profiles are:

- **DUN (dial-up networking)**. Supports links between, for example, a mobile phone and a laptop computer. This allows you to access the Internet by connecting to an Internet service provider through your mobile phone.
- **HID (human interface device)**. Supports Bluetooth-enabled HID-class devices, such as keyboards and mice. In most cases, this means that you can expect a Bluetooth-enabled HID-class device to work transparently with a Mac OS X system.

  Further, all Bluetooth-enabled HID-class devices are supported by the Mac OS X HID Manager. This means that you can use the HID Manager API to access your device.
- **Serial port**. Provides a bridge from the RFCOMM protocol to the built-in Mac OS X serial port driver. You can use this profile to support legacy applications that depend on direct serial-port access.
- **Object push**. Allows the transfer of small files (several hundred Kilobytes in size or less) between Bluetooth-enabled devices. You can use this profile to send and receive files in the vCard format, such as virtual business cards.
- **FTP (file transfer protocol)**. Allows a Bluetooth device to be treated as a remote file system. You can use the FTP profile to browse a remote Bluetooth device's file system, get directory listings, and transfer files.
- **Synchronization**. Supports synchronization of data between a computer and a device such as a Bluetooth-enabled PDA. You can use this profile to implement automatic data synchronization that occurs as soon as devices discover each other, rather than at a user's command.

With version 1.5 of the Bluetooth software, two new profiles became available:

- **HCRP (hardcopy cable replacement profile)**. This profile allows the transfer of rendered data between Bluetooth-enabled devices, such as between a laptop and a printer. It is assumed that the client device (in this case, the laptop) will include a driver that renders the data. Note that Mac OS X supports Bluetooth printing through the Mac OS X printing API.
- **Headset profile**. The headset profile allows an application to use a Bluetooth-enabled headset as the input or output audio device. After a headset is properly configured using the Bluetooth Setup Assistant application, an input and an output audio device associated with the headset are available for selection.

Mac OS X also provides a number of Bluetooth-specific applications. These applications guide users through various set-up procedures, such as configuring new Bluetooth devices and setting up serial-port communication.

In versions of Mac OS X prior to 10.4, the Bluetooth applications are:

- **Bluetooth File Exchange**. This application uses the FTP profile to support the exchange of files between two connected Bluetooth devices.
- **Bluetooth Serial Utility**. This application allows an advanced user to set up serial-port emulation. Bluetooth Serial Utility encapsulates the functionality that was available in the System Preferences in previous versions of Mac OS X.
- **Bluetooth Setup Assistant**. Using an easy, step-by-step approach, this application guides the user through the configuration of a new Bluetooth device, setting it up to work with system services, such as iSync.

In Mac OS X, version 10.4, Apple introduced two new Bluetooth applications:

- **Bluetooth Explorer**. This application allows you to:
  - Verify that your new Bluetooth service is properly registered
  - View a computer's Bluetooth hardware information
  - Perform inquiries and view detailed results regarding discovered devices
  - View active Bluetooth connections
  - Select different Bluetooth hardware attached to the computer (if more than one Bluetooth device is present on the computer)
- **Packet Logger**. This application monitors all Bluetooth traffic being transmitted on the computer and saves it to a log file. You can then view the captured data in the log file to help debug problems in your application, or with Bluetooth hardware.

These applications are available in `/Developer/Applications/Utilities/Bluetooth`. To use them, your computer must include a Bluetooth module.

Finally, you can use the Bluetooth preferences panel in System Preferences to place selected Bluetooth devices in categories, such as favorites. Note that the Bluetooth preferences panel appears in System Preferences only if a Bluetooth device is in range.

The Mac OS X Bluetooth API Overview—Two Frameworks

The Mac OS X Bluetooth API consists of two frameworks that provide all the methods and functions you need to access Bluetooth-specific functionality in your application:

- `IOBluetooth.framework`
- `IOBluetoothUI.framework`

Both frameworks are in `/System/Library/Frameworks` and each has a specific target.

The Bluetooth Framework

The **Bluetooth framework** contains the API you use to perform Bluetooth-specific tasks. With the methods and functions in the Bluetooth framework, you can:

- Create and destroy connections to remote devices
- Discover services on a remote device
- Perform data transfers over various channels
- Receive Bluetooth-specific status codes or messages

The Bluetooth UI Framework

The **Bluetooth UI framework** contains the API you use to provide a consistent user interface in your applications. This API provides Aqua-compliant panels your application can present to the user. These panels help the user to perform tasks such as creating connections, pairing with remote devices, and discovering services.

For maximum flexibility, the Bluetooth and Bluetooth UI framework APIs are available in both C and Objective-C. To emphasize the parity between the two versions, the APIs follow a naming convention that makes it easy to see the correspondence between entities. For example, methods in the Objective-C API refer to the object representing a Bluetooth device as IOBluetoothDevice. Functions in the C API refer to the same entity with an `IOBluetoothDeviceRef` variable.

The Bluetooth Classes

Whether you choose C or Objective-C to develop your application, it's important to become comfortable with an object-oriented perspective of Bluetooth connections. The Bluetooth framework defines object-oriented abstractions that encapsulate all the components in a Bluetooth connection. Channels, connections, and remote devices are all represented by objects. Even if you choose to develop a Bluetooth application in C, it helps to view the various entities in the connection as objects with specific, well-defined capabilities and responsibilities. This helps you visualize a Bluetooth connection and keep track of each object's scope of operation.

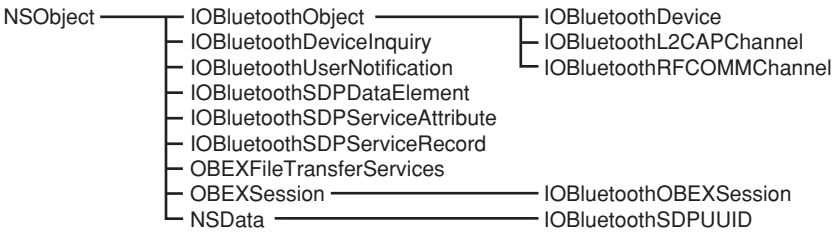
This section examines the Objective-C Bluetooth classes, describing their inheritance relationships and some of their functionality and displaying how they fit into the protocol stack. The next section, "Objects in Bluetooth Connections" (page 28), examines the objects instantiated from these classes and describes their dynamic relationships in a particular Bluetooth connection.

Note: Keep in mind that although this section uses Objective-C terminology to describe Bluetooth classes and objects, you can also access these entities in a C or C++ application.

The Bluetooth framework contains several classes, a few of which will never be handled directly by any application. Some of the classes are base classes that provide you with useful subclasses. Others are accessible only through instances that are created by intermediate objects. Nevertheless, an understanding of the class hierarchy will give you insight into the architecture of a Bluetooth connection in Mac OS X.

Figure 2-3 (page 25) shows the inheritance relationships of the Bluetooth framework classes.

Figure 2-3 The Mac OS X Bluetooth class hierarchy



As you can see in Figure 2-3 (page 25), the inheritance relationships among these classes are uncomplicated. All classes in the Bluetooth framework inherit directly or indirectly from NSObject, which is the root class of most Objective-C class hierarchies. NSObject provides its subclasses with the ability to behave as objects and with a basic interface to the run-time system for the Objective-C language.

IOBluetoothObject Class

The classes IOBluetoothDevice, IOBluetoothL2CAPChannel, and IOBluetoothRFCOMMChannel inherit from the **IOBluetoothObject class**. The IOBluetoothObject class imparts the ability to respond to Bluetooth-specific events. The **IOBluetoothDevice class** provides instance methods to open and close baseband connections to a remote device and to get information about that connection.

Mac OS X version 10.2.5 includes fully asynchronous versions of the IOBluetoothDevice methods to open and close L2CAP and RFCOMM channels. In Objective-C, these methods rely on the instantiation of a delegate object to receive notifications of incoming data and other callbacks. A delegate object is simply an object that performs a task on behalf of another object. For example, if an Objective-C application wants to receive notifications of the state of an L2CAP channel (such as open completed or closed) it can register its object as a delegate of the associated IOBluetoothL2CAPChannel object. The application then chooses to implement the delegate methods that relay information about the state of the channel. When the state of the L2CAP channel changes, the appropriate delegate method is called, notifying the application.

Because a C or C++ application does not have access to Objective-C delegates, it registers callback functions to receive notifications of the events it's interested in. The Bluetooth framework follows a naming convention that emphasizes the functional similarity of the delegate methods and events. For example, the Objective-C delegate method `l2capChannelOpenComplete:status:` is equivalent to the C event type `kIOBluetoothL2CAPChannelEventTypeOpenComplete`.

The **IOBluetoothL2CAPChannel class** and **IOBluetoothRFCOMMChannel class** provide methods to write data to the channels and to register for various open and close notifications. These two classes can also use a delegate as a target for data and events. Be aware that to take advantage of the benefits delegates provide, you must:

- Be running on Mac OS X version 10.2.5 or later
- Create a delegate object and register it as a client of the channel
- Implement at least the incoming data–callback method in your delegate

For more information on how to use the new asynchronous methods, see "Using Delegates to Receive Asynchronous Messages" (page 47).

OBEXSession Class

The **OBEXSession class** (which descends directly from NSObject) provides the methods required to manipulate an OBEX session. The OBEXSession class doesn't inherit from IOBluetoothObject, because the particular transport on which the connection is established is immaterial to the implementation of the OBEX protocol. With some work, you could use the OBEXSession class's methods to implement an OBEX session over a non-Bluetooth transport, such as Ethernet. The OBEXSession class is strictly concerned with the details of communicating using the OBEX protocol, regardless of the underlying transport.

The Bluetooth framework hierarchy includes one subclass of the OBEXSession class, the **IOBluetoothOBEXSession class**. This class provides methods to manipulate an OBEX session with a Bluetooth RFCOMM channel as the transport.

OBEXFileTransferServices Class

In Mac OS X version 10.4, Apple introduced the **OBEXFileTransferServices** class. Inheriting from NSObject, the new OBEXFileTransferServices class supports file-transfer operations beyond the `PUT` and `GET` primitives in the OBEX API. Using a valid IOBluetoothOBEXSession object, you create an OBEXFileTransferServices object which you can use to perform FTP or object push operations.

The OBEXFileTransferServices API includes delegate methods that correspond to the class's connection, disconnection, and folder manipulation instance methods.

IOBluetoothUserNotification Class

The **IOBluetoothUserNotification class** encapsulates a user notification registered on a Mac OS X system. When your application registers for certain notifications (such as incoming channel notifications), it receives an IOBluetoothUserNotification object. The single method the IOBluetoothUserNotification class provides allows you to unregister from these notifications.

Remaining Classes

In Mac OS X version 10.4, Apple introduced the **IOBluetoothDeviceInquiry** class. A subclass of NSObject, the IOBluetoothDeviceInquiry class is intended for the small subset of applications that cannot use the GUI-based device-discovery API provided in the Bluetooth UI framework. Although the Bluetooth specification describes an inquiry process, there are good reasons not to implement it without restrictions (for more information on this, see "Inquiring and Paging" (page 38)). The IOBluetoothDeviceInquiry class provides methods that allow applications to perform Bluetooth inquiries while enforcing limitations that ensure the best possible user experience.

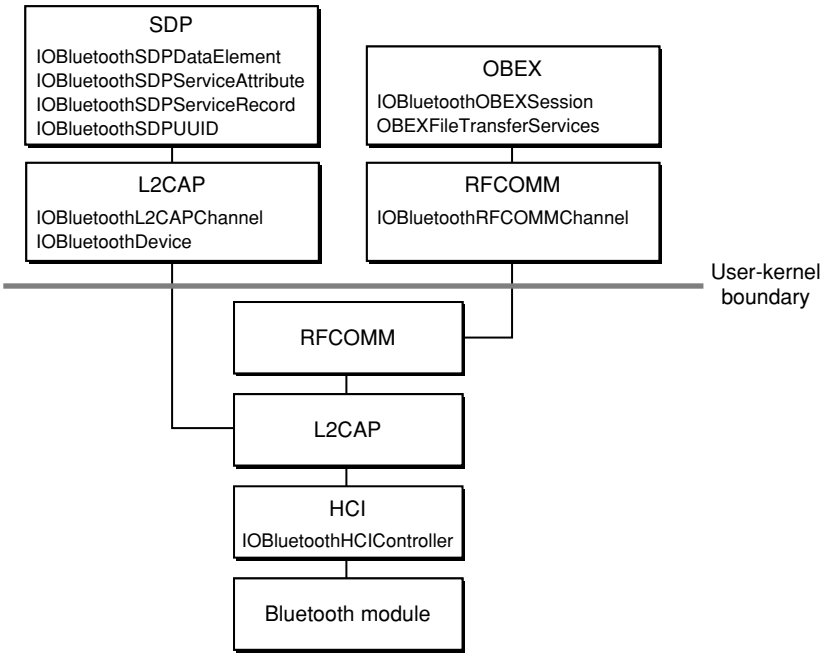
The remaining classes shown in Figure 2-3 (page 25) are concerned with the implementation of the service discovery protocol. Recall that information about the services a Bluetooth device offers is contained in that device's SDP database. The SDP database comprises a set of records, each of which uses service attributes to describe the service. The **IOBluetoothSDPServiceRecord class** provides instance methods to get information about individual SDP service records. As its name suggests, the IOBluetoothSDPServiceAttribute class provides methods to create, initialize, and get information about the attributes of SDP service records.

The information contained in each service attribute is a value of a particular type (such as Boolean value or text string) with a particular size. A device must be prepared for the type and size of the information it receives in a service attribute, so the attributes are sent inside SDP data elements. An SDP data element is part type descriptor and part size descriptor. The **IOBluetoothSDPDataElement class** maps the data types described in the Bluetooth specification onto various Foundation classes. These classes include NSNumber, NSString, and NSArray, as well as the NSData subclass, **IOBluetoothSDPUUID**, which provides methods to represent and manipulate UUIDs. In addition, the IOBluetoothSDPDataElement class provides methods to create, initialize, and get information about the data element of each service attribute. (NSNumber, NSArray, and NSData are subclasses of NSObject; for more information on these and other Foundation classes, see the Foundation API reference at Cocoa Documentation.)

Bluetooth Classes in the Mac OS X Bluetooth Protocol Stack

The inheritance relationships pictured in Figure 2-3 (page 25) display the hierarchical structure of the Bluetooth framework classes. This helps describe their spheres of influence. Figure 2-4 (page 28) shows the classes that applications can use (in addition to the IOBluetoothHCIController class you can subclass to support Bluetooth over an alternate transport) and how they fit into the protocol stack.

Figure 2-4 Bluetooth classes in the Bluetooth protocol stack



The next section describes the objects that are instantiated from these classes (with the exception of the IOBluetoothHCIController class) and how they work together to represent a connection to a remote Bluetooth device.

Objects in Bluetooth Connections

You can think of a connection to a remote Bluetooth device as a vertical slice through the Bluetooth protocol stack (shown in Figure 2-1 (page 20)). Depending on precisely which protocols are involved, the connection is built of pieces from the appropriate layers. View these pieces as objects and you can see how a Bluetooth connection might be modeled.

This section describes the roles of the Bluetooth framework objects in Bluetooth connections. Bear in mind that many of these objects are automatically created by other objects. When a Bluetooth-enabled device is discovered, low-level drivers and higher-level managers create objects to represent it. Most applications, therefore, never need to create directly many of these objects to perform their functions.

Note: Remember that Bluetooth objects are also accessible from C and C++ applications using references of the form `ObjectNameRef`.

For more information on how to use these objects in a Bluetooth application, see "Developing Bluetooth Applications" (page 35).

The Root Object

The **IOBluetoothDevice object** is the root object of every Bluetooth connection. You use this root object to create connections to the device, including the initial baseband connection. For example, you can open L2CAP and RFCOMM channels using methods of the IOBluetoothDevice object. You can also use its methods to perform SDP service searches.

The IOBluetoothDevice object is the only object in the Bluetooth stack that can be created without a concrete connection behind it. In other words, it can exist even before you create the baseband connection to the device. You might, for example, know a device's address from an earlier device search. Using this device address, you can instantiate an IOBluetoothDevice object to represent the device. You can then access cached information about the device, such as device name and SDP attributes, while deferring the creation of the baseband connection until the user selects that device.

In Mac OS X version 10.2.4 and later, the IOBluetoothDevice object includes API to support device categorization based on user-selected criteria. The new methods and functions allow you "mark" a device as a recently accessed device or as a user-designated favorite. Your application can use these new device attributes to display to the user only those devices in a specific category. For more information on the use of these attributes, see "Filtering and Validation" (page 32).

Objects as Data Conduits

The **IOBluetoothL2CAPChannel object** represents the data conduit between a local and a remote device and is a required component in every Bluetooth connection. This is because the L2CAP layer provides the facilities higher-layer protocols need to communicate across a Bluetooth link. An IOBluetoothL2CAPChannel object provides methods to read from and write to the channel. It also provides a `setDelegate:` method that allows a client to register itself as a client of the L2CAP channel.

An IOBluetoothL2CAPChannel object can also exist in the absence of a concrete channel. Unlike an IOBluetoothDevice object, however, it can be created only when an IOBluetoothDevice object opens an L2CAP channel. An IOBluetoothL2CAPChannel object can persist after the closure of its associated L2CAP channel, but it will return errors for any calls it receives.

The **IOBluetoothRFCOMMChannel object** represents an RFCOMM channel. Like the IOBluetoothL2CAPChannel object, the IOBluetoothRFCOMMChannel object provides API to open and close the channel and read from and write to the channel. In addition, an RFCOMMChannel object makes available methods to receive event notifications.

Objects in OBEX Connections

An OBEXSession object represents an OBEX connection to a remote device. Because it handles the commands typical of an OBEX session, the object itself is agnostic about the type of transport underlying the connection. The object's usefulness lies in the methods it provides the object of a transport-specific subclass, such as the RFCOMM-based IOBluetoothOBEXSession class. Thus, your Bluetooth application will never create or access a raw OBEXSession object. Instead, it will get an OBEXSession subclass object, like an IOBluetoothOBEXSession object. With this object, it can override some of the OBEXSession class's methods to manipulate the OBEX session.

The **IOBluetoothOBEXSession object** represents an OBEX session using a Bluetooth RFCOMM channel as the transport. If your application acts as an OBEX server or client over an RFCOMM channel, you first create an IOBluetoothOBEXSession object. Then, you use its methods (and some of its superclass's) to open and close the transport connection and send and receive data.

Beginning in Mac OS X version 10.4, if your application needs to perform FTP or object push operations over OBEX, you can use a valid IOBluetoothOBEXSession object to create an **OBEXFileTransferServices** object. Note that the IOBluetoothOBEXSession does not have to be connected to a remote server to successfully create an OBEXFileTransferServices object. The connection can be made explicitly, using one of the OBEXFileTransferServices connection methods or implicitly, when you call any of its file-transfer methods.

A Device-Discovery Object

Although most applications can perform discovery of in-range devices using the APIs in the Bluetooth UI framework, some must perform non-GUI inquiries. If your application is running in Mac OS X version 10.4 or later, you can use the IOBluetoothDeviceInquiry object to find devices in proximity to the computer and, optionally, retrieve name information from them.

It's important to realize that the device inquiry process, although it is supported by the Bluetooth specification, can negatively affect other devices in the vicinity. For this reason, you are encouraged to use the GUI-based device discovery methods the Bluetooth UI framework provides. For more information on the problems that can arise from device inquiries and pages, see "Inquiring and Paging" (page 38).

If your application cannot use the GUI-based device discovery methods, you can use the methods of the IOBluetoothDeviceInquiry object. Before you use this object, however, it is essential that you are aware of both its built-in restrictions and the limits your application must observe when using it. For specific information on using the IOBluetoothDeviceInquiry object, see "Performing Device Inquiries" (page 47).

Objects Related to Service Discovery

There are four objects that support SDP-related tasks:

- IOBluetoothSDPServiceRecord
- IOBluetoothSDPServiceAttribute
- IOBluetoothSDPDataElement
- IOBluetoothSDPUUID

The **IOBluetoothSDPServiceRecord object** represents a service offered by a Bluetooth device. A Bluetooth device can make several services available; each is described by an instance of IOBluetoothSDPServiceRecord in its SDP database. Each IOBluetoothSDPServiceRecord object contains two elements that, together, describe the service:

- A reference to the device offering the service.
- An NSDictionary of service attributes.

When an SDP client (a device that is looking for a particular service) performs an SDP query, it looks for a service record that contains one or more specific attributes.

The **IOBluetoothSDPServiceAttribute object** represents a single SDP service attribute. An IOBluetoothSDPServiceRecord object can contain several IOBluetoothSDPServiceAttribute objects in its instance of NSDictionary. Each IOBluetoothSDPServiceAttribute object contains an attribute ID (an unsigned 16-bit integer) and an instance of IOBluetoothSDPDataElement that describes one of the service's attributes. The Bluetooth specification defines a large number of attributes a profile can use to describe a service, in addition to those the profile defines. Examples of attributes are the service's name, list of supported Bluetooth profiles, and service ID.

An **IOBluetoothSDPDataElement object** represents a single SDP data element. To alert a device to what type and size of data it is about to receive in a data element, the IOBluetoothSDPDataElement object provides this information along with the data itself. An IOBluetoothSDPDataElement object contains:

- A data element type descriptor
- A data element size descriptor (calculated from the actual size of the data element)
- The size of the data element
- The data element value itself

The Bluetooth specification defines nine types of data elements, including unsigned integer, URL, and text string. Apple has mapped these types onto appropriate Foundation classes such as NSNumber and NSString. In addition, Apple has defined the class **IOBluetoothSDPUUID** to describe the UUID (universally unique ID) data element type. An IOBluetoothSDPUUID object can represent a UUID of any valid size (16, 32, or 128 bits) and contains methods to compare UUIDs and convert a smaller UUID to a larger one.

Because Mac OS X automatically creates IOBluetoothSDPDataElement objects for most client and server operations, your application should never need to explicitly create them. The exception to this is if you plan to define a new SDP service and make it available to the system. In this case, you will appreciate Apple's decision to use Foundation classes to represent both the collection of service attributes and the types of data elements.

Instead of writing a lot of code to build up a service record, you can define your new service with a collection of key-value pairs in a property list (`plist`) file. You can then import this file into your application, using the `IOBluetoothAddServiceDict` function to create a new IOBluetoothSDPServiceRecord object. The `IOBluetoothAddServiceDict` function uses the properties in your `plist` file to populate the IOBluetoothSDPServiceRecord's NSDictionary of service attributes. For more information on how to do this, see "Providing a New Service" (page 40).

The Bluetooth UI Classes

The Bluetooth UI framework provides objects, methods, and functions your application can use to perform user interface tasks. As with the API of the Bluetooth framework, the Bluetooth UI framework contains Objective-C and C versions of its API. The primary objects in the API are subclasses of NSWindowController, which supports the loading, displaying, and closing of windows, among other things.

By using the user interface objects defined in the Bluetooth UI API, you can obtain a consistent look and feel across all your Bluetooth applications—without having to write code to implement standard user interface tasks, such as searching for devices or services and creating paired-device relationships.

Common to both the C and Objective-C versions of the API are the following three objects:

- **Service browser controller**. In the Objective-C API, an IOBluetoothServiceBrowserController object and in the C API, an `IOBluetoothServiceBrowserControllerRef`

  This object displays a window in which a user can find in-range Bluetooth devices, perform SDP queries on them, and select SDP services.

- **Device selector controller**. In the Objective-C API, an IOBluetoothDevice SelectorController object and in the C API, an `IOBluetoothDeviceSelectorControllerRef`

  This object displays a window in which a user can select a particular device with which to communicate.

- **Pairing controller**. In the Objective-C API, an IOBluetoothPairingController object and in the C API, an `IOBluetoothPairingControllerRef`

  This object displays a window in which a user can initiate pairing with a remote Bluetooth device. If necessary, this object will also prompt the user for a personal identification number (PIN) to complete the pairing process.

Filtering and Validation

In Mac OS X version 10.2.4, the Bluetooth UI framework includes support for filtering and validation in its user interface objects. The service browser controller, device selector controller, and pairing controller objects allow the user to filter the list of available devices and services by the following categories:

- Device type
- Favorite devices
- Recently accessed devices

Not only does filtering narrow the list of available devices and services presented to the user, it also makes it easier for the user to connect to particular devices. For example, if a user marks a device as a favorite, it is listed as such in all user interface panels the Bluetooth UI framework provides. The user can then select this favorite device, usually with a single click, bypassing the device discovery process. This is especially useful in an environment filled with Bluetooth devices.

Filtering is a user-initiated process, designed to make it easier for users to find and connect to Bluetooth devices. Validation, on the other hand, is application-initiated. It is designed to ensure that the user can select only those devices that support services the application designates. To perform validation, the user interface objects available in Mac OS X version 10.2.4 and later perform an SDP query on the device the user chooses. Before finalizing the user's selection, the user interface object verifies that the device does indeed offer the service the application desires. If it does, the application is guaranteed to have access to the appropriate service. If it doesn't, the user is prompted to make an alternate selection.

With service validation a part of the selection process, user experience is enhanced. Your application can quickly inform the user if the selected device is appropriate or not, while still in the context of the selection process. Otherwise, your application must accept the user's selection, perform the validation, and repeat the selection process if the selection is invalid.

Display Options for User Interface Panels

Also introduced in Mac OS X version 10.2.4 is the option to run a user interface panel as a sheet on a target window or in a modal session.

A sheet is a dialog panel that's attached to its associated window so that a user never loses track of which window the dialog belongs to. While a sheet is open, the user is prevented from doing anything else in the window that owns the sheet until the sheet is dismissed. When the sheet is dismissed, an application-defined delegate object receives the results of the user's action.

By running a panel in a modal session, an application can perform lengthy operations while still sending events to the panel. If you choose to run a user interface panel in a modal session, the user interface object will validate the user's selection before the method returns.

In addition, all text in a user interface panel can be customized and localized, regardless of display mode.

Developing Bluetooth Applications

This chapter describes how to develop Bluetooth applications for Mac OS X. In this context, *Bluetooth applications* encompasses the full range of applications that access Bluetooth-enabled devices, directly or indirectly. Whether you're interested in vending a Bluetooth service or in making sure your application handles a Bluetooth device just like any other, this chapter will help get you started.

This chapter first presents an overview of different types of Bluetooth applications. Then it discusses general design principles to consider while developing a Bluetooth application for Mac OS X. Finally, it describes some specific tasks an application might need to perform.

Overview of Bluetooth Application Types

Bluetooth applications run the gamut from games that can utilize a Bluetooth input device to applications that vend Bluetooth services or support new profiles. Different applications need different levels of "Bluetooth awareness" to successfully perform their functions. Some applications may be able to use a high-level manager that Mac OS X provides without ever having to use the Bluetooth API. Others will use the Bluetooth API extensively to provide Bluetooth-specific services. The integration of Bluetooth support on Mac OS X supports applications throughout this range.

To help you decide at what level your application needs to communicate with a Bluetooth device, this section surveys some typical actions a Bluetooth application might take. After you read this section, you'll have a better idea of what parts of the Mac OS X Bluetooth API (if any) you need to use.

Accessing a HID-Class Device

If you're writing a game or other application that accepts input from a HID-class device, you may be wondering if you need to do something special to support Bluetooth-enabled devices. Alternatively, if you provide a Bluetooth-enabled HID-class device, you might wonder how to implement special features in your driver. Fortunately, Apple has done most of the work for you. In Mac OS X version 10.2.5 and later, Apple provides a fully compliant HID-class driver that supports Bluetooth. This means that Bluetooth-enabled, HID-class devices work transparently on Mac OS X version 10.2.5 and later: Your game or other application that accepts input from a HID-class device need not be concerned with the device's transport. Equally as important, it also means that you do not have to use Apple's Bluetooth API to access such devices.

When you configure a HID-class device, the Apple-provided HID-class driver loads and takes care of all the protocols and profiles that talk to the Bluetooth module on the device. You do not need to write a kernel-resident driver to gain access to the device. Instead, your application can access and even customize a HID-class device using Apple's HID Manager client API. The Mac OS X HID Manager client API provides access to a HID-class device through a device interface exported to user space by the HID family. For example, using the HID Manager client API you can:

- Open or close a device
- Get the most recent value of an element
- Set an element value

For more information on using the HID Manager client API to access a HID-class device, see *Working With HID Class Device Interfaces*. For sample code illustrating how to use the HID Manager client API, see Games Human Interface Device & Force Feedback Sample Code.

Although you don't need to use the Bluetooth API to access a HID-class device, you may choose to use functions or methods from the Bluetooth framework to enhance the user's experience. For example, your application can provide Bluetooth-specific information that lets the user know if a device doesn't support a particular service. You can read about some of these tasks in "A Collection of Specific Tasks" (page 40).

Accessing Serial Ports

The Bluetooth serial port profile forms the basis of a number of other profiles, such as dial-up networking, generic object exchange, and object push. In addition, the serial port profile provides a serial port emulation layer that supports applications that require direct serial port access. With the Bluetooth serial port profile, these applications can treat a Bluetooth link as a serial cable link using standard POSIX ttys.

In general, applications that depend on direct serial port access are legacy applications or, perhaps, the debugging components of other applications. Even though it may seem easier for legacy applications to use the serial port emulation layer exclusively when communicating with Bluetooth devices, there are drawbacks:

- All Bluetooth-specific errors are reported as a failure to open the serial port.

  Whether, for example, the Bluetooth device could not be found or it does not support the desired service, the user is informed that the serial port could not be opened. This is frustrating for the user because it does not accurately describe the problem and provides no guidance on how to fix it.

- The user must set up the serial port.

  A legacy serial port application requires the user to set up the serial port. This is not a trivial task and can be confusing for novice users.

As an alternative to using the serial port profile, Apple strongly recommends that legacy applications expecting direct serial port access be updated to use the Bluetooth RFCOMM API. Using the RFCOMM API, you get the best of both worlds:

- Unfettered access to the serial ports
- Complete control over the creation, behavior, and destruction of RFCOMM channels

- Fine-grained error reporting, including the reporting of Bluetooth-specific errors and status messages
- A clean and comprehensive user interface featuring integrated Bluetooth UI panels for device selection

Vending a Bluetooth Service

Bluetooth support on Mac OS X version 10.2 and later allows you to create new services in software and make them available to remote clients. Using the APIs in the Bluetooth frameworks, you define a service, ensure that it is visible to others, and serve it to remote clients.

Of course, which Bluetooth APIs you use depends on the nature of the service you plan to offer. One task that is common to all such applications, however, is the addition of the service to the local SDP database. A service must be present in the SDP database so remote clients can find it during an SDP inquiry. Apple has streamlined this task by defining:

- The scope of the service
- The format of the service's attributes

A service's scope can be either transient or persistent. A transient service exists only while the application that registered it is running. When that application closes, the service is automatically removed and the Mac OS X Bluetooth system performs any necessary clean up. As its name suggests, a persistent service persists beyond the running of the application that registered it; it even persists across reboots. A persistent service can initiate the launch of a client application when a remote connection requests the service.

As described in "Objects in Bluetooth Connections" (page 28), the Bluetooth system uses a dictionary format to define a Bluetooth service. In a service's dictionary, each entry corresponds to a service attribute. This scheme makes a new service easy to define because you can describe it in a property list. It also makes it easy to add the dictionary to the SDP database. Using the Bluetooth framework API you can create an SDP service record from your dictionary, add it to the SDP database, and remove it from the database.

For specific examples detailing how to work with service-attribute dictionaries, see "Providing a New Service" (page 40).

Another task specific to creating a new service to vend is getting a UUID to identify it. The Bluetooth specification defines UUIDs for various profiles and services. In addition, the SDP describes a method of generating UUIDs that guarantees an extremely small chance of duplication. For more information on the basis of this method, see http://www.opengroup.org/publications/catalog/c706.htm. Apple uses Core Foundation functions to generate UUIDs. For more information on how to do this, see "Generating a UUID" (page 43).

General Design Considerations

This section discusses general design considerations you should keep in mind as you develop a Bluetooth application for Mac OS X. Presented in no particular order, these considerations will help you produce an application that is easy to use and takes full advantage of Bluetooth technology.

Inquiring and Paging

The Bluetooth specification describes an inquiry process designed to find all Bluetooth devices in the immediate area. The process consists of sending out frequent inquiries and waiting for acknowledgements from in-range devices. The Bluetooth specification also defines a paging process in which a device or application sends out frequent pages to a particular device. If the target device is in range (and willing to connect) it will send a positive response to the page.

Although the Bluetooth specification supports the inquiry and paging processes, the practical implementation of these processes can lead to some undesirable effects, such as:

- An application that is constantly performing inquiries is disrupting 802.11b traffic in the vicinity.

  This can severely degrade a system's or device's ability to send and receive other Bluetooth communication. In addition, it needlessly pollutes the wireless environment, adversely affecting other 802.11b devices.

- Performing frequent pages does not result in a positive user experience.

  One of the main reasons an application would want to perform device paging is to simulate a device-proximity detector: When the desired device comes into range, it responds to the application's page and triggers some specific task in the application. The paging process is a lengthy one, however, and like the inquiry process, results in degraded Bluetooth communication for its duration. In the worst case (when the desired device is not present), the timeout for the page could be as much as 15 seconds.

- For a device to be visible to an inquiry, it must be in discoverable mode.

  The vast majority of devices are not in discoverable mode by default. The user must actively choose to make a device discoverable. In addition, a device that is always in discoverable mode is using more power. Since most Bluetooth-enabled devices are wireless, this means a greater drain on the battery. Finally, a perpetually discoverable device is more vulnerable to unwanted connections.

The alternative to the discoverable mode is the connectable mode. When a device is connectable, it doesn't respond to inquiries but it does respond to specific connection requests. Apple's Bluetooth UI framework provides device and service discovery methods and functions that find known devices and avoid the problems listed above.

In Mac OS X version 10.4, Apple introduced the IOBluetoothDeviceInquiry class. Using the IOBluetoothDeviceInquiry object instantiated from this class, your application can perform non-GUI device inquiries.

Note: You are strongly encouraged to use the device-discovery methods provided in the Bluetooth UI framework. Only if your application absolutely cannot use them should you consider using the IOBluetoothDeviceInquiry methods instead. If you do use the IOBluetoothDeviceInquiry API, be sure you heed the warnings described in "Performing Device Inquiries" (page 47).

Bandwidth Constraints

Bluetooth was developed as a low-bandwidth, wireless connectivity solution. It's important to keep the bandwidth constraint in mind when designing your application. The total per-link budget for throughput and bandwidth is 720 kbps. This means that there are 720 kbps available to be shared among *all* connections on a link. When an application assumes that all 720 kbps are available to a single connection, there are two negative consequences:

- The performance on other connections is degraded
- The application is likely to experience a much lower level of throughput, especially if the user has selected to use a Bluetooth mouse or keyboard. Quality-of-service constraints require Mac OS X to devote bandwidth to these devices, as well.

It's important to scrutinize your application's bandwidth and throughput needs. If you do require a full 720 kbps of bandwidth, Bluetooth is probably the wrong choice for your wireless connectivity.

SCO

As described in "The Bluetooth Protocol Stack" (page 12), SCO stands for synchronous, connection-oriented links. These links are used primarily for voice communication. With Bluetooth 1.5 (which runs in Mac OS X version 10.3.2 and later), Apple introduced support for the headset profile, which is based on a SCO link. Using a computer equipped with an internal Apple Bluetooth module (or a D-Link DBT-120 rev. B or later) running the latest firmware, you can use a Bluetooth-enabled headset to communicate using iChat AV 2.1 public beta or later.

It's important to realize that, in its current implementation, SCO is not adequate for speech recognition in Mac OS X. Although the operating system could support it, most SCO-based headsets would not be able to deliver the 22 kHz, 16-bit resolution required for speech recognition.

Device Interfaces

If you're familiar with the I/O Kit, Apple's object-oriented framework for developing device drivers, you may also be familiar with the concept of a device interface. A device interface is an I/O Kit construct that allows you to access hardware from applications. Using a device interface, you can develop an application-based device driver that enjoys the same level of control available to in-kernel drivers.

You do *not* need to use a device interface to access a Bluetooth device from an application. The Bluetooth framework APIs available in Mac OS X version 10.2 and later provide everything you need to access both Bluetooth devices and objects in the Bluetooth protocol stack. Using the Bluetooth framework APIs, you can:

- Create and destroy connections
- Receive notifications of device appearance and disappearance
- Transfer data to and from a device

A Collection of Specific Tasks

Every Bluetooth application is unique, but there are a number of tasks that are common to many applications. This section presents several of these tasks and describes how to perform them using the Bluetooth and Bluetooth UI API.

Much of the code illustrating these tasks is drawn from the sample applications available in `/Developer/Examples/Bluetooth` on Mac OS X version 10.2 and later. The samples are also included in the SDK, available at http://developer.apple.com/hardware/bluetooth/. Examine these sample applications to see how the individual tasks in this section fit into a complete application.

Providing a New Service

If you're providing a new Bluetooth service, you must make that service available through the local SDP database. This ensures that your service is visible to potential clients performing SDP service searches. Then, you wait for a client to request your service.

There are five steps in providing a service:

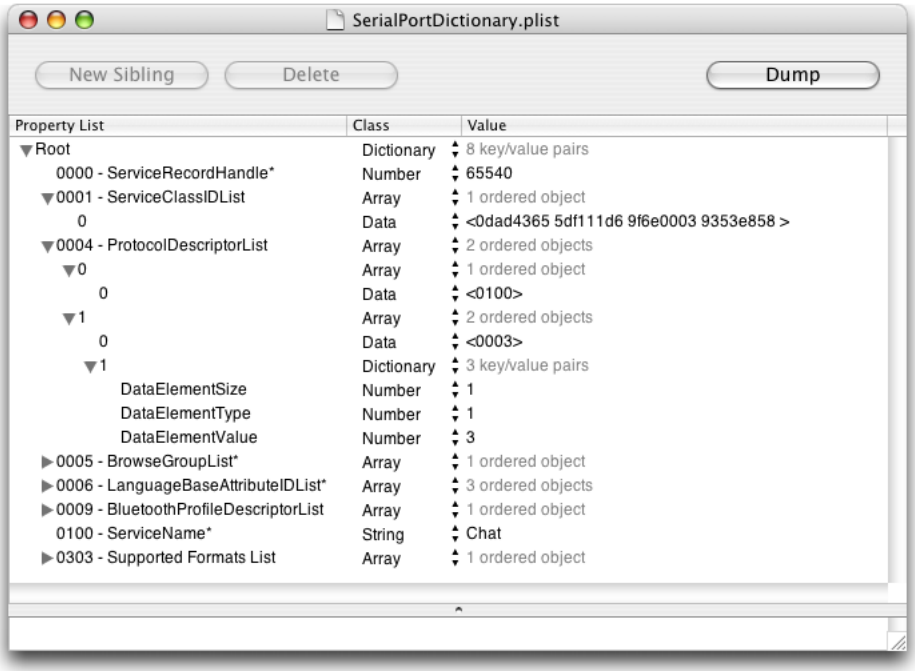
1. Define the service.
2. Generate a UUID for the service (or, if you're providing a predefined service, use the UUID defined in the profile).
3. Add the service definition to the SDP database.
4. Register for notification of the opening of the incoming channel assigned to the service.
5. When finished providing the service, remove the service definition from the SDP database.

The following sections describe these five steps in detail. API reference for the functions in this section is available at Device Drivers Bluetooth Documentation or on disk from `/Developer/Documentation/DeviceDriversIOKit/DeviceDriversIOKit.html`.

Defining a Service

As described in "Objects in Bluetooth Connections" (page 28), you define a service by creating a dictionary in which each key-value pair, or property, corresponds to a service attribute. Apple makes it easy to add new services to the system by supporting the importation of a `plist` file that contains the dictionary. Thus, instead of building up a potentially complex dictionary in code, you can use the Property List Editor application to create it. Then, you use the Bluetooth API to load the dictionary into your application. Figure 3-1 (page 41) shows a portion of the dictionary that describes the RFCOMM Chat Server service.

Figure 3-1 Partial listing of RFCOMM Chat Server service dictionary



Each property in the dictionary corresponds to one of the many service attributes defined by the Bluetooth specification (or to one of the attributes defined in the service's profile). The property's key is the attribute ID, and the value is the attribute's data value. For example, the first property in the dictionary in Figure 3-1 (page 41) describes the service record handle, a 32-bit number that uniquely identifies the service within the server. The third key-value pair describes the Bluetooth protocols the service needs.

Each attribute key is a string that must begin with a hexadecimal number representing the attribute's ID. These IDs are defined in the file `BluetoothAssignedNumbers.h`, available in the Bluetooth framework. If you choose, you can add other characters to the key string, but only after a space following the hexadecimal ID number. For example, each key in the dictionary above displays the name of the attribute after the ID number and a space.

Each attribute value contains the information that describes the attribute. The Bluetooth specification defines several data types that identify the different types of data the attribute values can contain. For example, an unsigned integer is type 1, and a sequence of data elements is type 6. Apple has mapped these data types onto Foundation classes such as NSNumber and NSArray. In turn, these classes correspond to native property list types, such as `integer` and `array`. This chain of correspondence makes it easy to translate a dictionary in a `plist` file into a service record object that represents your service.

As you can see in Figure 3-1 (page 41), however, the attribute values differ significantly from one another. This is because different attribute-value types can be described in different ways. Formally, an attribute value is described by a combination of three components:

- The size of the data
- A description of the data type
- The data itself

This set of information is neatly captured in a three-property dictionary in which each property key names a component and each corresponding value holds the information. An example of such a dictionary is inside the value of the protocol descriptor list key shown in Figure 3-1 (page 41):

```
<dict>
    <key>DataElementSize</key>
    <integer>1</integer>
    <key>DataElementType</key>
    <integer>1</integer>
    <key>DataElementValue</key>
    <integer>3</integer>
</dict>
```

However, to make it easier to create an attribute dictionary in a `plist` file, Apple provides some shortcuts for common data types:

- If the attribute's value is of type `string`, `data`, or `array`, you do not need to provide a data-element size property. This is because the Mac OS X Bluetooth system will infer the data type from the dictionary type and calculate the size from the data itself.
- If the attribute's value is a string or an unsigned, 32-bit integer, no three-property dictionary is required to describe it. The value of the service record handle key in Figure 3-1 (page 41) is an example of an unsigned, 32-bit integer value.
- If the value is of the `nil` type (type 0), neither the data-element size property nor the data-element value property is required.
- If the data type is either unsigned integer or signed two's complement integer, you can use the `data` property list type to hold the value. In this case, the numeric data is read into the service record in network-byte order (most significant byte first). No data-element size property is needed for these data types when you use the `data` property list type.
- If the attribute's value is a UUID, you can use the `data` property list type to hold it. The Mac OS X Bluetooth system will infer the data type and calculate the size from the value.
- If the attribute's value is a list, such as the protocol-descriptor list attribute in Figure 3-1 (page 41), you use the `array` property list type to represent it. Each array member describes a member of the list. Because each array member is itself a data element, it must conform to the guidelines for attribute values.

At this time, there are two service-attribute properties you do not have to place in your service dictionary:

- Service record handle
- RFCOMM channel ID

Because there is a single name space for service record handles and channel IDs, the Bluetooth system assigns these when your application imports the dictionary. If you do include these properties in your service-attribute dictionary, the Bluetooth system attempts to use them. If the values you specify are already in use, the Bluetooth system assigns others instead.

The service attributes discussed so far are defined by the Bluetooth specification. They are made available through the service-discovery process so potential clients can make informed choices. In addition to these attributes, Apple defines optional local attributes that control the local behavior of the service. These attributes are not visible to remote clients. At this time, Apple defines two local attributes:

- Persistent
- Target application

You specify these attributes in a special property with the key `LocalAttributes`, placed at the root level of your service's attribute dictionary. The value of the `LocalAttributes` key must be a dictionary whose members are individual local attributes.

The persistent attribute (identified by the key `Persistent`) accepts a Boolean type. A value of TRUE indicates that the service should persist beyond the application that initiated it and across system reboots. When you use this attribute to designate a service as persistent, the service-record handle is automatically saved. It is used to restore the service whenever the associated Bluetooth hardware is present. It's essential that your application save its own copy of a persistent service's record handle, too. This is because the only way to programmatically remove a persistent service is to pass the record handle to the `IOBluetoothRemoveServiceWithRecordHandle` function. (For information on how to force the removal of a service whose handle you don't know, see "Removing a Service Without a Handle" (page 46).)

By default, the absence of the `Persistent` attribute causes the service to be transient. (Note that the absence of the local attributes property as a whole also means the service is considered transient.) This means that the service is automatically removed when the client application terminates. If you choose, you can remove a transient service before the client application exits by calling `IOBluetoothRemoveServiceWithRecordHandle`.

The second local attribute specifies an application that should be launched when a remote device attempts to connect to the service. This happens when a remote device tries to open an L2CAP or RFCOMM channel of the type specified in the service's service record. The key of this local attribute is `TargetApplication`, and the value must be a string containing the absolute path to the target application's executable file. If no `TargetApplication` attribute is present (or if the local attributes dictionary is absent), no special action is taken when a remote device connects to the service. In this case, it's the application's responsibility to watch for the connection and take the appropriate steps.

Generating a UUID

To generate a UUID for your service, you can use the command-line utility `uuidgen`. Simply type `uuidgen` on the command line to receive a unique 128-bit value in the form of a hyphen-punctuated ASCII string, as in this example:

```
% uuidgen
4302FA6D-089B-11D8-96C4-0030656F08FE
```

You then use this UUID to identify your service. In the unlikely event you need a new UUID each time your code executes, you can use a Core Foundation function to generate it. Listing Listing 3-1 (page 43) shows how to do this.

Listing 3-1 Generating a new UUID in code

```
#include <CoreFoundation/CoreFoundation.h>
```

```
int main()
{
    CFUUIDRef   uuid;
    CFStringRef string;

    uuid = CFUUIDCreate( NULL );
    string = CFUUIDCreateString( NULL, uuid );

    CFShow( string );
}
```

Adding a Service Definition to the SDP Database

After you've described your service's attributes in a `plist` file, you're ready to use it in your application. The following is an outline of the steps you take to make your service available:

1. Create a dictionary and initialize it with the contents of your `plist` file.
2. Create an SDP service record that contains the attributes in your dictionary.
3. Preserve the newly created service-record handle. This is required if your service is persistent or if you plan to terminate a transient service before your application closes.
4. If necessary, preserve the RFCOMM or L2CAP channel the system assigns to your service.

The code in Listing 3-2 (page 44) shows how to implement these steps. It assumes that you already defined an attribute dictionary in a `plist` file and know the file's path. For brevity's sake, only limited error handling is shown.

Listing 3-2 Making a new service available

```
- (BOOL)publishService
{
    NSString            *dictionaryPath = nil;
    NSString            *serviceName = nil;
    NSMutableDictionary *sdpEntries = nil;

    // Create a string with the new service name.
    serviceName = [NSString stringWithFormat:@"%@ My New Service", [self
                           localDeviceName]];

    // Get the path for the dictionary we wish to publish.
    dictionaryPath = [[NSBundle mainBundle]
                pathForResource:@"MyServiceDictionary" ofType:@"plist"];

    if ( ( dictionaryPath != nil ) && ( serviceName != nil ) )
    {
        // Initialize sdpEntries with the dictionary from the path.
        sdpEntries = [NSMutableDictionary
                    dictionaryWithContentsOfFile:dictionaryPath];

        if ( sdpEntries != nil )
        {
            IOBluetoothSDPServiceRecordRef  serviceRecordRef;
```

```
            [sdpEntries setObject:serviceName forKey:@"0100 - ServiceName*"];

        // Create a new IOBluetoothSDPServiceRecord that includes both
        // the attributes in the dictionary and the attributes the
        // system assigns. Add this service record to the SDP database.
            if (IOBluetoothAddServiceDict( (CFDictionaryRef) sdpEntries,
                        &serviceRecordRef ) == kIOReturnSuccess)
            {
                IOBluetoothSDPServiceRecord *serviceRecord;

                serviceRecord = [IOBluetoothSDPServiceRecord
                            withSDPServiceRecordRef:serviceRecordRef];

                // Preserve the RFCOMM channel assigned to this service.
                // A header file contains the following declaration:
                // IOBluetoothRFCOMMChannelID mServerChannelID;
                [serviceRecord getRFCOMMChannelID:&mServerChannelID];

                // Preserve the service-record handle assigned to this
                // service.
                // A header file contains the following declaration:
                // IOBluetoothSDPServiceRecordHandle mServerHandle;
                [serviceRecord getServiceRecordHandle:&mServerHandle];

                // Now that we have an IOBluetoothSDPServiceRecord object,
                // we no longer need the IOBluetoothSDPServiceRecordRef.
                IOBluetoothObjectRelease( serviceRecordRef );

            }
        }
    }
}
```

Getting Channel-Open Notifications

The system assigns a particular RFCOMM channel to your service, and your application needs to know when a client is opening that channel. To do this, you register for a channel-open notification. Listing 3-3 (page 45) shows how to do this, using the RFCOMM channel ID you saved when you added your service dictionary (as shown in Listing 3-2 (page 44)).

Listing 3-3 Registering for a channel-open notification

```
// Register for a notification so we get notified when a client opens
// the channel assigned to our new service.
// A header file contains the following declaration:
// IOBluetoothUserNotification *mIncomingChannelNotification;

mIncomingChannelNotification = [IOBluetoothRFCOMMChannel
        registerForChannelOpenNotifications:self
        selector:@selector(newRFCOMMChannelOpened:channel:)
        withChannelID:mServerChannelID
        direction:kIOBluetoothUserNotificationChannelDirectionIncoming];
```

Withdrawing a Service

When your application is ready to stop providing your service, you must remove it from the SDP database so it is no longer available to potential clients. To do this, you use the service record handle you saved when you first added your service dictionary to the database. In addition, you should unregister for the channel open notifications for which you registered earlier. Listing 3-4 (page 46) shows how to perform these tasks.

Listing 3-4 Preparing to stop providing a service

```
- (void)stopProvidingService
{
    if ( mServerHandle != 0 )
    {
        // Remove the service.
        IOBluetoothRemoveServiceWithRecordHandle( mServerHandle );
    }

    // Unregister the notification.
    if ( mIncomingChannelNotification != nil )
    {
        [mIncomingChannelNotification unregister];
        mIncomingChannelNotification = nil;
    }

    mServerChannelID = 0;
}
```

Removing a Service Without a Handle

If you've created a persistent service and your application crashes before you're able to save the service-record handle, you can remove it manually, as the following steps show.

Note: Following these steps results in the loss of all cached Bluetooth information on your system, such as device names and paired-device information. This should only be done as a last resort and *never* by an end user.

1. Using NetInfo Manager, enable the root user.
2. Open the Terminal application and log in as root.
3. Change directory to `/var/root/Library/Preferences`.
4. Remove the `blued.plist` file.

Important: The `blued.plist` file is an internal implementation detail. You should not write code that depends on its location or existence.

Using Delegates to Receive Asynchronous Messages

Beginning in Mac OS X version 10.2.5, you can use delegates in your Objective-C application to receive asynchronous messages sent by L2CAP and RFCOMM channels. These messages include notifications of incoming data and channel status changes. When an L2CAP or RFCOMM channel is opened, the client of the channel uses the `setDelegate:` method to designate a delegate. It's often convenient for the client to make itself the delegate, as in this example:

```
[newRFCOMMChannel setDelegate:self]
```

If you choose to employ a delegate to receive asynchronous messages, you must implement at least the incoming data delegate method. Other delegate methods, such as those that receive channel status messages, are optional. The header files `IOBluetoothL2CAPChannel.h` and `IOBluetoothRFCOMMChannel.h` (both located in the Bluetooth framework) define informal protocols that describe the available delegate methods. For example, as a client of an RFCOMM channel, in addition to the `rfcommChannelData:data:length:` method, you can implement any of the following delegate methods:

- `rfcommChannelOpenComplete:status:`
- `rfcommChannelClose:`
- `rfcommChannelControlSignalChanged:`
- `rfcommChannelFlowControlChanged:`
- `rfcommChannelWriteComplete:refcon:status:`
- `rfcommChannelQueueSpaceAvailable:`

Performing Device Inquiries

The IOBluetoothDeviceInquiry class is designed to allow Bluetooth-supported inquiries while avoiding the worst of the negative consequences associated with the device inquiry process. The class does this by limiting the amount of time spent on inquiries within a certain time period.

Your application, too, must bear some of the responsibility for the smooth operation of the device inquiry process. In particular:

- You should not try to circumvent the restriction of inquiries by calling `start` on the IOBluetoothDeviceInquiry object multiple times in quick succession. After you call `start`, the inquiry may take several seconds to begin and calling `start` many times in a row does not change this. Therefore, you should call `start` only once to begin an inquiry process, recognizing that the inquiry may not begin as soon as you expect it to. If you implement the `deviceInquiryStarted` delegate method, you will be able to tell when the inquiry process has begun to search for devices.
- You can shorten the length of time the IOBluetoothDeviceInquiry object spends on the inquiry process, but you should still not initiate multiple inquiries within that length of time.

- You must not initiate your own remote device-name requests while this object is performing a device inquiry or from the delegate methods you implement. If you do this, you could deadlock your process.

  If you need to perform your own name requests on remote devices, do so *only* after you have stopped the IOBluetoothDeviceInquiry object.

You can tell the IOBluetoothDeviceInquiry object to perform name requests on the remote devices it finds by calling the `setUpdateNewDeviceNames` method. You can then retrieve the information after your `deviceInquiryDeviceFound` delegate method is invoked.

By default, the IOBluetoothDeviceInquiry object performs the broadest possible inquiry, searching for devices of any major and minor device class that report any major service class. If you choose, you can use the `setSearchCriteria` method to restrict the inquiry process to consider only devices that report a specific service class or that belong to a specific major or minor device class. The `BluetoothAssignedNumbers.h` header file (located in the Bluetooth framework) lists the device and service class values you can use. It is not recommended that you do this, however, because not all devices identify themselves or their services in a standard manner. If you perform a restricted device inquiry, you can miss devices you might be interested in.

Document Revision History

This table describes the changes to *Working With Bluetooth Devices*.

Date	Notes
2004-06-29	Updated for Mac OS X version 10.4.
2004-06-28	Added information about the headset profile.
2004-05-27	Minor change to Bluetooth protocol stack illustration.
2004-01-22	Added information about the hardcopy cable replacement profile (HCRP).
2003-11-15	First version.

Document Revision History