Albert S. Huang • Larry Rudolph

Bluetooth

ESSENTIALS FOR PROGRAMMERS

CAMBRIDGE www.cambridge.org/9780521703758

This page intentionally left blank

Bluetooth Essentials for Programmers

This book provides an introduction to Bluetooth programming, with a specific focus on developing real code. The authors discuss the major concepts and techniques involved in Bluetooth programming, with special emphasis on how they relate to other networking technologies. They provide specific descriptions and examples for creating applications in a number of programming languages and environments, including Python, C, Java, GNU/Linux, Windows XP, Symbian Series 60, and Mac OS X. No previous experience with Bluetooth is assumed, and the material is suitable for anyone with some programming background. The book places special emphasis on the Python language examples, showing the translation of concepts and techniques into actual working programs. Programmers who have never seen or used Python before will find these examples easy to follow and understand.

Albert S. Huang is a PhD candidate at MIT's Computer Science and Artificial Intelligence Laboratory (CSAIL). His primary areas of research are robotics and computer vision.

Larry Rudolph, PhD, received his doctorate from NYU's Courant Institute. He is currently a Principal Research Scientist at MIT and a cofaculty member at the New England Complex Science Institute. His most recent research has been in the field of mobile computing.

Bluetooth Essentials for Programmers

Albert S. Huang and Larry Rudolph
Massachusetts Institute of Technology

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK
Published in the United States of America by Cambridge University Press, New York
www.cambridge.org
Information on this title: www.cambridge.org/9780521703758

First published in print format 2007

ISBN-13 978-0-511-35583-7 eBook (NetLibrary)
ISBN-10 0-511-35583-1 eBook (NetLibrary)

ISBN-13 978-0-521-70375-8 paperback
ISBN-10 0-521-70375-1 paperback

Contents

Preface

About This Book

This book provides an introduction to Bluetooth wireless technology* and Bluetooth programming, with a specific focus on the parts of Bluetooth that concern a software developer. While there is already a host of existing literature about Bluetooth, few of these texts are written for the programmer who is concerned primarily with creating Bluetooth software applications. Instead, they tell all about Bluetooth, when most of the time, the programmer is interested only in a tiny fraction of this information.

This book purposefully and happily leaves out a great deal of information about Bluetooth. Concepts are simplified and described in ways that make sense to a programmer, not necessarily the ways they're laid out in the Bluetooth specification. The approach is to start simply, allowing the reader to quickly master the basic concepts with the default parameters before addressing a few advanced features.

Despite these omissions, this book is a rigorous introduction to Bluetooth, albeit with a narrow focus. Applications can be developed without an understanding of the radio modulation techniques or the algorithms underlying the generation of Bluetooth encryption keys. Programmers, however,

* Throughout this book, we will abbreviate the phrase "Bluetooth wireless technology" with the concise word "Bluetooth."

do need to understand issues such as the available transport protocols, the processes governing establishing connections, and the mechanisms for transferring data.

We strongly believe in learning by example and have included working programs that demonstrate the concepts and techniques introduced in the text.* Examples are provided for a wide variety of programming languages and environments, including Python, C, and Java, running on GNU/Linux, Windows, Nokia Series 60, and OS X. Special emphasis is given to the Python language examples – the simplicity and clarity of Python allows us to very easily show the translation of concepts and techniques into actual working programs. We believe that programmers who have never seen or used Python will find these examples easy to follow and understand.

This book is not meant to be a be-all and end-all guide to Bluetooth programming; rather it is meant to serve as a stepping stone, the first foothold for programmers interested in working with Bluetooth. The exposition of concepts and demonstration of techniques should be sufficient to allow any programmer to start creating his or her own functional Bluetooth applications that can interoperate with many other Bluetooth devices. For those interested in a deeper understanding of the inner workings and nitty-gritty details of Bluetooth, this book serves as sufficient preparation to enable one to tackle the more complex and technical documents like the Bluetooth specification itself.

Audience

This book targets the computer programmer looking for an introduction to Bluetooth and how to program with it. It assumes no previous knowledge of Bluetooth (you may have never even heard of it before picking up this book), but does assume that you have some programming experience and have access to and can use either a GNU/Linux, Windows XP, or OS X development environment.

Because Bluetooth programming shares much in common with network programming, there will be frequent references and comparisons to concepts in network programming, such as sockets and the TCP/IP transport protocols. A basic understanding of these concepts will help solidify your understanding

* Visit our Web site to download the examples in the book. Check updates, errata, and new material: www.btessentials.com or www.cambridge.org/9780521703758.

of Bluetooth programming, but the text is written such that it does not assume the reader has this knowledge.

Organization of This Book

Chapter 1 provides an introduction to Bluetooth and the essentials of Bluetooth programming. The first chapter is divided into two major sections. The first section can be considered the bare essentials, most of which must be understood in order to create a functional Bluetooth program. The second section is material that is highly relevant to the Bluetooth programmer, but is not of critical importance when creating simple Bluetooth programs.

Chapter 2 shows how the concepts and techniques introduced in Chapter 1 can be implemented in the Python programming language using the PyBluez extension module. The simplest examples (Device Discovery, Service Search, and RFCOMM) work on both GNU/Linux and Windows XP SP2 computers, and a few examples require a GNU/Linux computer. This chapter allows the programmer to quickly and easily create fully functional Bluetooth programs with minimal effort.

Chapter 3 describes in detail how to create Bluetooth applications targeted for the GNU/Linux operating system. Programming examples are given in C, and require the BlueZ stack, which is now a part of the standard Linux kernel. Device Discovery, Service Search, RFCOMM, L2CAP, HCI, and SCO are described.

Chapter 4 describes the current state of Bluetooth programming in Windows XP. A detailed introduction is then given for the Microsoft Bluetooth stack, which comes standard in Windows XP SP2 (and is expected to be present in Windows Vista as well). Examples are given for Device Discovery, Service Search, and RFCOMM.

Chapter 5 describes how to create Bluetooth applications using the Java programming language. JSR-82, the standard Java API for Bluetooth programming, is introduced along with examples for Device Discovery, Service Search, RFCOMM, and L2CAP.

Chapter 6 provides a partial introduction to Bluetooth programming on the Nokia Series 60 and OS X operating systems. The Series 60 Python Bluetooth API is described with examples for Device Discovery, Service Search, and RFCOMM. Device Discovery and outgoing RFCOMM connections using the OS X Objective C Bluetooth API are introduced with examples. Lastly, Bluetooth serial ports are introduced as an alternative to standard

Bluetooth programming. Bluetooth programming using RFCOMM serial ports is described along with its merits and drawbacks. Examples are given for GNU/Linux, Windows XP, and OS X.

Chapter 7 introduces a series of GNU/Linux Bluetooth development tools that are indispensable for the serious Bluetooth developer. Although these tools are built only for the GNU/Linux operating system, Bluetooth programmers creating applications targeted for other platforms may still find these tools highly useful.

Acknowledgments

A large number of people made valuable contributions to this text. Marcell Holtmann and the BlueZ developers provided outstanding support throughout our trials and tribulations with Bluetooth. Simson Garfinkel and Yotam Gingold provided advice on the OS X section. Sally Lee has been absolutely wonderful with her assistance on the production of the text. Angela Hsieh provided the cover artwork ^.^ Heather Bergman, Pooja Jain, and the rest of Cambridge University Press have been fantastic with their guidance and support. Kari Pulli, Jamey Hicks, and Max Van Kleek provided numerous bits of insight and advice. The following students in the Oxygen Research Group have also been of great help: Ning Song, Calvin On, Emily Yan, Atish Nigam, Sonia Garg, Debbie Wan, Michael D'Ambrosio, Nancy Kho, Xiao Yu, and Jessica Hwang. Finally, Sally Lee deserves lots of thanks for her helpful advice in nearly all aspects of this work.

We would also like to thank the MIT Project Oxygen, Nokia, and the Singapore–MIT alliance for their support of the work that eventually led to the writing of this book.

Albert's Personal Comments

This book grew out of my master's thesis work with Larry on Bluetooth and location-aware computing. During that time, it was difficult to find good documentation and introductory material suitable for someone with a computer science background. I started writing what I figured out and in a way that I would have loved to have had when I first started out. Like many endeavours, we weren't sure how it would turn out and if anybody would actually read

it. But my hope is that this book will help lower the entry bar for working with Bluetooth.

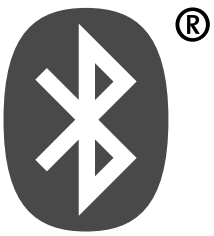
All of this would not have been possible without the love and support of my family, friends, and colleagues. Most of all, I'd like to thank Stacy, my parents, my sister Heather, and Jen. Thank you for everything.

Larry's Personal Comments

A book is a large undertaking and in the case of this book, it was like trying to fit a square block into a round pinhole. All of my time has been allocated to family, teaching, advising, and research, but somehow time was found for this enterprise. Albert did the bulk of the work and deserves the bulk of the credit. I merely managed the work. Nevertheless, because of this project, I spent less time with my family and so would like to give my thanks to Hilla, Noga, and Amit, as well as to Ainat.

The need for a programming guide became evident over a half a dozen years ago when Sonia Garg valiantly struggled to control Bluetooth devices in a nonstandard way as part of our research. Without her persistence, it is likely that I would have avoided any future attempt to deal with Bluetooth. The MIT Project Oxygen got me involved in pervasive, human-centric computing, and deserves a lot of thanks. In addition, I would like to thank my friends at Nokia and the Nokia Research Center. Finally, I want to thank CSAIL, the Computer Science and Artificial Intelligence Laboratory, at MIT.

Chapter 1



Introduction

Bluetooth is *a way for devices to wirelessly communicate over short distances*. Wireless communication has been around since the late nineteenth century, and has taken form in radio, infrared, television, and more recently 802.11. What distinguishes Bluetooth is its special attention to short-distance communication, usually less than 30 ft. Both hardware and software are affected by this special attention.

A comprehensive set of documents, called the *Bluetooth Specifications*,* describes, in gory detail, everything from the basic radio signal to the high-level protocols for this wireless, short-range communication. Our goal is to help you master the essentials and then with a pat on the back and a firm handshake, let you wade through the gory details on your own, confident that you are well prepared. In other words, our goal is to help the reader get started.

1.1 Understanding Bluetooth as a Software Developer

There is nothing especially difficult about Bluetooth, although the unusually wide scope of Bluetooth makes it hard to get started. There is so much in the Bluetooth specification and at so many different levels, it is hard to distinguish the essentials from the elective. Technology specifications, especially ones

* http://www.bluetooth.com/Bluetooth/Learn/Technology/Specifications/

that are given folksy names, often refer to something very specific and with a narrow scope. For example, there is no single specification of the Internet. Rather each of the components has its own specification. Ethernet, for example, describes how to connect a bunch of machines together to form a simple network, but that's about it. TCP/IP describes two specific communication protocols that form the basis of the Internet, but they're just two protocols. Similarly, HTTP is the basis behind the World Wide Web, but also boils down to a simple protocol. The Internet is a collection of these and other pieces. A software developer first learning about Internet programming – how to connect one computer on the Internet to another and send data back and forth – will not initially bother with the details of Ethernet or e-mail, precisely because neither technology is central. Sure, e-mail is a wonderful example of Internet programming and Ethernet gives context on how the connections are implemented, but TCP/IP programming is the essence.

In many ways, the word Bluetooth is like the word Internet because it encompasses a wide range of subjects. Similar to USB, Ethernet, and 802.11, Bluetooth defines a lot of physical on-the-wire stuff, such as the radio frequencies and how to modulate and demodulate signals. Similar to Voice-over-IP protocols used in many Internet applications, Bluetooth also describes how to transmit audio between devices. But Bluetooth also specifies everything in between! It's no wonder that the Bluetooth specifications are thousands upon thousands of pages.

This text answers the question: How do we create programs that connect two Bluetooth devices and transfer data between them? We introduce the essential concepts, as well as application design considerations. Later chapters show how to actually do this with a number of popular programming languages and operating systems. Most importantly, we keep it simple but sufficient.

1.2 Essential Bluetooth Programming Concepts

The essentials of Bluetooth programming are neither numerous nor difficult. Throughout the rest of this chapter, they will be compared to those of Internet programming. Although Bluetooth was designed from the ground up, presumably independent of the Ethernet and TCP/IP protocols, it is quite reasonable to think of Bluetooth programming in the same way as Internet programming. Both fall under the general rubric of network programming,

and share the same principles of one device communicating and exchanging data with another device. Bluetooth and Internet programming share so much in common that understanding one makes it much easier to understand the other.

TCP/IP programming is mature, ubiquitous, has plenty of examples, and its comparison with Bluetooth strengthens the reader's understanding of both topics. The biggest difference, as mentioned earlier, is that Bluetooth focuses on physically proximate devices, whereas Internet programming doesn't care about distance at all. This difference will greatly affect how two devices initially find each other and establish an initial connection. After that, everything is pretty much the same.

The actual process of establishing a connection depends on whether the device in question is establishing an *outgoing* or an *incoming* connection. Roughly, this is the difference between which device sends the first data packet to initiate communications and which device receives that packet. We'll often refer to these as the *client* and *server*, respectively.

> *Note:* We use the words *client* and *server* only to distinguish between which device initiates a connection, and without implying any relationship to the Client–Server model of network programming. Despite the overlap, it is perfectly reasonable for a server (the way we use the word) to function as a client (in the Client–Server model sense), and vice versa.

Devices initiating an outgoing connection need to choose a target device and a transport protocol, before establishing a connection and transferring data. Devices establishing an incoming connection need to choose a transport protocol, and then listen before accepting a connection and transferring data. Figures 1.1 and 1.2 illustrate these basic concepts and how they translate to both Internet and Bluetooth programming. Notice that for outgoing connections, only the first two steps (choosing a target device, transport protocol, and port number) are different for Bluetooth and Internet programming. Once the outgoing connection is established, everything is pretty much the same. The processes for accepting an incoming connection are even more similar, with the main difference being the added support of Bluetooth for dynamically assigned port numbers.

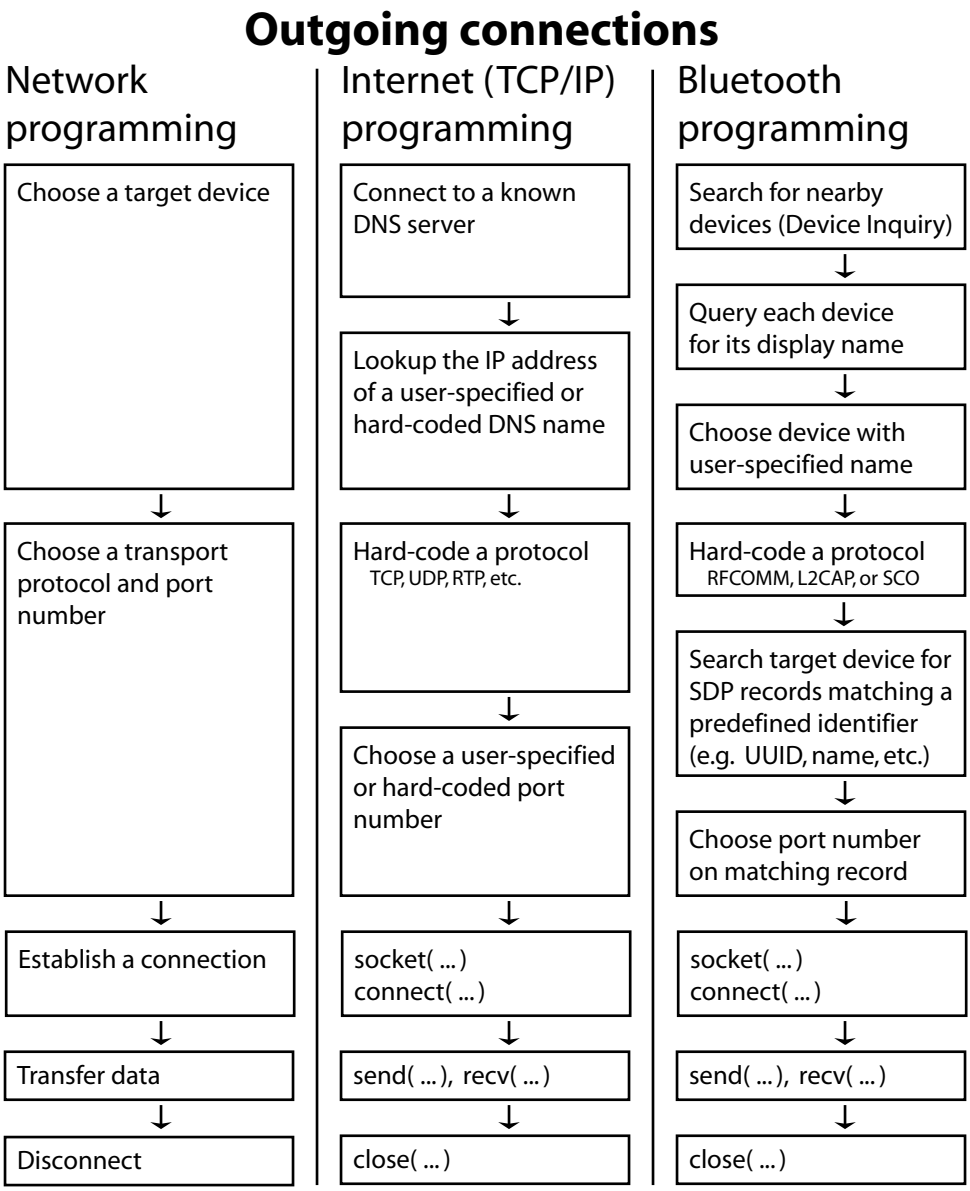


Figure 1.1 There are five major conecptual steps for programming an outgoing connection. Only the initial process of choosing a target device, transport protocol, and port number differs significantly from Internet to Bluetooth programming.

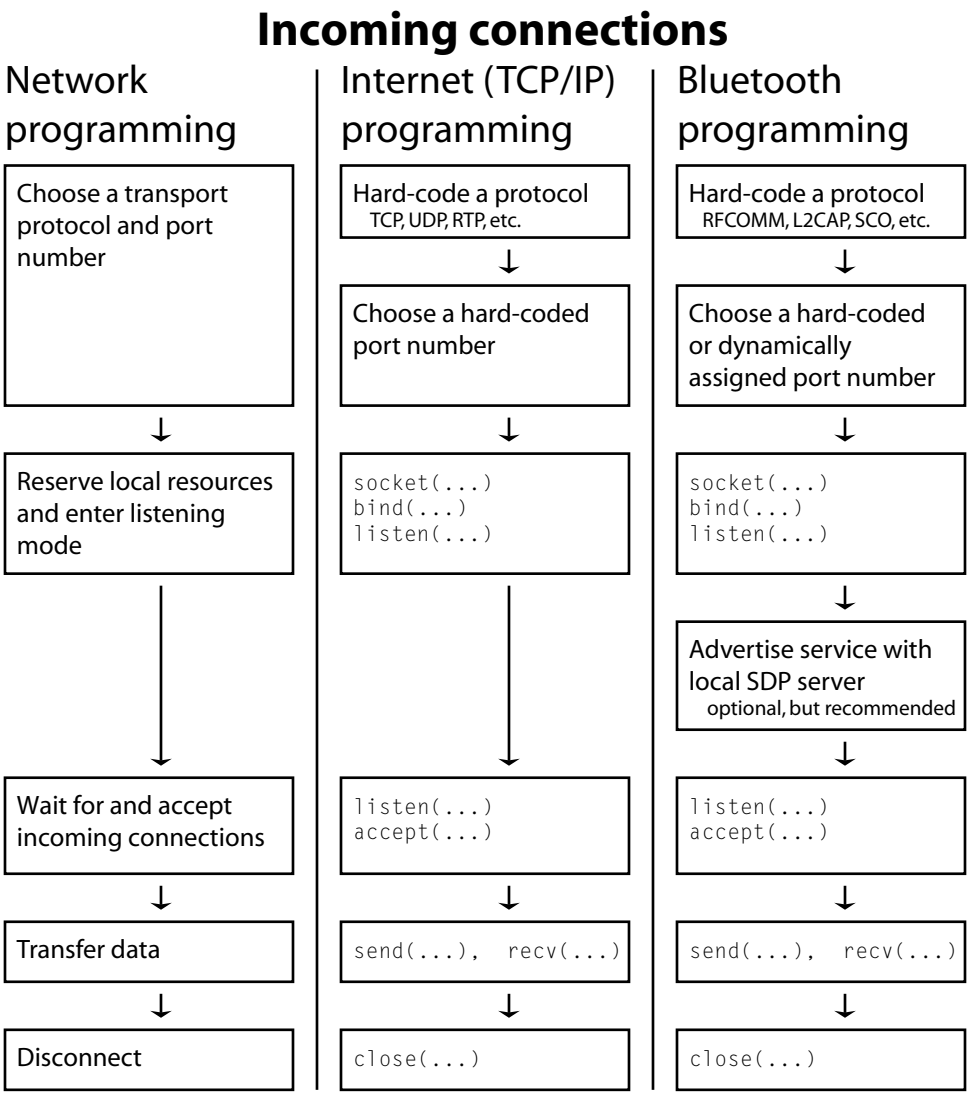


Figure 1.2 There are also five major conecptual steps for programming an incoming connection. As with outgoing connections, the details of Internet and Bluetooth programming are related, although slightly different.

The seasoned network programmer will notice that some of the steps illustrated don't always apply, and don't necessarily have to be completed in the order shown. For example, if the address of a server is hard-coded into a client program, then there's no need to use the Domain Name System (DNS) or a device inquiry. The key here is that these diagrams show the "vanilla" way of doing things, which can be adapted and tweaked to serve the needs of each individual application. As we go through each concept, in the next several sections, we mention each of these deviations in turn. Subsequent chapters elaborate on how these concepts are implemented across various platforms and programming languages.

1.2.1 Choosing a Target Device

Every Bluetooth chip ever manufactured is imprinted with a globally unique 48-bit address, referred to as the *Bluetooth address* or *device address*. This is identical in nature to the Machine Address Code (MAC) address for Ethernet.* In fact, both address spaces are managed by the same organization – the IEEE Registration Authority. These Bluetooth addresses, assigned at manufacture time, are intended to be unique and remain static for the lifetime of the chip. It conveniently serves as the basic addressing unit in all of Bluetooth programming.

For one Bluetooth device to communicate with another, it must have some way of determining the other device's Bluetooth address. The address is used in all layers of the Bluetooth communication process, from the low-level radio protocols to the higher level application protocols. In contrast, TCP/IP network devices that use Ethernet discard the 48-bit MAC address at higher layers of the communication process and switch to using IP addresses. The principle remains the same however, in that the unique identifying address of the target device must be known in order to communicate with it.

The client program may not have advance knowledge of these target addresses. In Internet programming, the user typically knows or supplies a host name, such as `www.kernel.org`, which must then be translated to a physical IP address by DNS. In Bluetooth, the user will typically supply some user-friendly name, such as "My Phone," and the client translates this to a numerical address by searching nearby Bluetooth devices and checking the name of each device.

* `http://www.ietf.org/rfc/rfc0826.txt`

> *Note:* Throughout the book, we will use the words *adapter* and *device*, with slightly different meanings. In general, a Bluetooth *device* refers to any machine capable of Bluetooth communication. When we're talking about writing programs, *adapter* refers specifically to the Bluetooth device on the local computer (the one that's running the program). This is to reduce confusion and help specify exactly which device is in question.

Device Name

Humans do not deal well with 48-bit numbers, such as `0x000EED3D1829` (in much the same way, we do not deal well with numerical IP addresses like 68.15.34.115), and so Bluetooth devices will almost always have a *user-friendly name* (also called the *display name*). This name is usually shown to the user in lieu of the Bluetooth address to identify a device, but ultimately it is the Bluetooth address that is used in actual communication. For many machines, such as mobile cell phones and desktop computers, this name is configurable and the user can choose an arbitrary word or phrase. There is no requirement for the user to choose a unique name, which can sometimes cause confusion when many nearby devices have the same name. When sending a file to someone's phone, for example, the user may be faced with the task of choosing from five different phones, each of which is named "My Phone."

Names in Bluetooth are similar to DNS names in that both are human-friendly identifiers that eventually get translated to machine-friendly identifiers. They differ in that DNS names are unique (there can only be one `www.google.com`) and registered with a central database, whereas Bluetooth names are more or less arbitrary, frequently duplicated, and are registered only on the device itself. In TCP/IP, one begins with a DNS name. Translating a DNS name to an IP address involves contacting a nameserver, issuing a query, and waiting for a result. In Bluetooth, the lookup process is reversed. First, a device searches for nearby Bluetooth devices and compiles a list of their addresses. Then, it contacts each nearby device individually and queries it for its user-friendly name.

Searching for Nearby Devices

Device discovery, also known as device inquiry, is the process of searching for and detecting nearby Bluetooth devices. It is simple in theory: To figure

out what's nearby, broadcast a "discovery" message and wait for replies. Each reply consists of the address of the responding device and an integer identifying the general class of the device (e.g., cell phone, desktop PC, headset, etc.). More detailed information, such as the name of a device, can then be obtained by contacting each device individually.*

In practice, this is often a confusing and irritating subject for Bluetooth developers and users. The source of this aggravation stems from the fact that it can take a long time to detect nearby Bluetooth devices. To be specific, given a Bluetooth cell phone and a Bluetooth laptop sitting next to each other on a desk, it will usually take an average of 5 seconds before the phone detects the presence of the laptop, and it sometimes can take upward of 10–15 seconds. This might not seem like that much time, but put in context it is suprising. During the device discovery process, the phone is changing frequencies more than a thousand times a second, and there are only 79 possible frequencies on which it can transmit. It is a wonder why they don't find each other in the blink of an eye.

The technical reasons for this are mostly due to the result of a strangely designed search algorithm, explained in greater detail in Section 1.3.9. Newer versions of Bluetooth (starting in Bluetooth 1.2) attempt to reduce the average search time, but don't expect any miracles in the near future. Suffice to say, device discovery takes too long.

> *Note:* A common misconception is that when a Bluetooth device enters an area, it somehow "announces" its presence so that other devices will know that it's around. This never happens (even though it's not a bad idea), and the *only* way for one device to detect the presence of another is to conduct a device discovery.

Discoverability and Connectability

For privacy and power concerns, all Bluetooth devices have two options that determine whether or not the device responds to device inquiries and connection attempts. The *Inquiry Scan* option controls the former, and the

* Bluetooth 2.1 introduces the *Extended Inquiry Response*, where the most commonly requested information, such as the name of a responding device and a summary of the services it offers, can be transmitted directly in the inquiry response, saving some time.

Table 1.1 Inquiry Scan and Page Scan.

Inquiry Scan	Page Scan	Interpretation
On	On	The local device is detectable by other Bluetooth devices, and will accept incoming connection requests. This is often the default setting.
Off	On	Although not detectable by other Bluetooth devices, the local device still responds to connection requests by devices that already have its Bluetooth address. This is often the default setting.
On	Off	The local device is detectable by other Bluetooth devices, but it will not accept any incoming connections. This is mostly useless.
Off	Off	The local device is not detectable by other Bluetooth devices, and will not accept any incoming connections. This could be useful if the local device will only establish outgoing connections.

Page Scan option controls the latter. Both are described in Table 1.1, and are configurable to some degree on most devices.

Although the names *inquiry scan* and *page scan* are confusing, it is important not to confuse these two terms with the actual processes of detecting and connecting to other devices. The reasoning behind these names is that these options control whether the local device *scans* for device inquiries and connection attempts. A device is in discoverable mode when inquiry scan is on.

1.2.2 Choosing a Transport Protocol

Different applications have different needs, hence the reason for different transport protocols. This section describes the ones you should know about for Bluetooth programming, and why your application might use them.

The two factors that distinguish the protocols here are *guarantees* and *semantics*. The guarantees of a protocol state how hard it tries to deliver a packet sent by the application. A *reliable* protocol, like TCP, takes the "deliver-or-die" mentality; it ensures that all sent packets get delivered, or dies trying (terminates the connection). A *best-effort* protocol, like UDP, makes a reasonable attempt at delivering transmitted packets, but ignores the case

when the packets do not arrive, say, due to signal noise, a crashed router, an act of god, and so on.

The semantics of a protocol can be either packet-based or streams-based. A packet-based protocol like UDP sends data in individual datagrams of fixed maximum length. A streams-based protcol, like TCP, on the other hand, just sends data without worrying about where one packet ends and the next one starts. This choice is more a matter of preference than any specific requirements, because with a little arm-twisting, a packet-based protocol can be used like a streams-based protocol, and vice versa.

Bluetooth contains many transport protocols, nearly all of which are special purpose. We consider four protocols to be essential, although only two, RFCOMM and L2CAP, are likely to be used to get started. And of these two, only the first, RFCOMM, will be relevant for many readers. The protocols are presented in the order of their relevance; the anxious reader can skim over the latter two or three.

RFCOMM

The Radio Frequency Communications (RFCOMM) protocol is a reliable streams-based protocol. It provides roughly the same service and reliability guarantees as TCP. The Bluetooth specification states that it was designed to emulate RS-232 serial ports (to make it easier for manufacturers to add Bluetooth capabilities to their existing serial port devices), but we prefer to turn that definition around and say that *RFCOMM is a general-purpose transport protocol that happens to work well for emulating serial ports*.

The choice of port numbers is the biggest difference between TCP and RFCOMM from a network programmer's perspective. Whereas TCP supports up to 65,535 open ports on a single machine, RFCOMM allows only 30. This has a significant impact on how to choose port numbers for server applications.

A distinguishing attribute of RFCOMM is that, depending on the application and the target platform, sometimes it is the only choice. Some environments, such as the Microsoft Windows XP Bluetooth API and Nokia Series 60 Python, support only the RFCOMM transport protocol. This really isn't all that bad because it's a robust general-purpose protocol, but it is something worth keeping in mind if you were to consider the other protocols mentioned. For more information, see Section 1.3.10.

L2CAP

The Logical Link Control and Adaption Protocol (L2CAP) is a packet-based protocol that can be configured with varying levels of reliability.* The default maximum packet size is 672 bytes, but this can be negotiated up to 65,535 bytes after a connection is established.

L2CAP can be compared with UDP, which is a best-effort packet-based protocol, but there are enough differences that the use cases for L2CAP are much broader than the use cases for UDP. Both are packet-based protocols, but L2CAP enforces delivery order. In UDP, it is possible for a computer to transmit two packets and have the second packet arrive before the first, due to quirks in Internet routing. This never happens in L2CAP, and packets are always delivered in the order they were sent. Additionally, UDP is restricted to best-effort guarantees, whereas L2CAP can be configured for varying levels of reliability. These differences mean that L2CAP can often be used where UDP would be used, for simple best-effort packet-based communication, but it can also be used for applications where UDP would not be suitable.

Reliability is acheived by a transmit/acknowledgment scheme in which unacknowledged packets may or may not be retransmitted. There are three possible retransmit policies:

- never retransmit (but make a best effort);
- always retransmit until success or total connection failure (reliable, the default); and
- drop a packet and move on to queued data if a packet hasn't been acknowledged after a specified time limit (0–1279 ms). This is useful when data must be transmitted in a timely manner (and it assumes a best effort).

These policies often seem useful at first glance, but there are some major pitfalls to watch out for. The first is that adjusting the delivery semantics for a single L2CAP connection to a remote device affects *all* L2CAP connections to that device. L2CAP connections to other devices are not affected. Second, L2CAP actually serves as the transport protocol for RFCOMM, so every RFCOMM connection is actually encapsulated within an L2CAP connection. This fact, combined with the first, means that changing

* This is accurate for all intents and purposes, but is not technically correct. See the next section (ACL) for details.

the delivery guarantees for L2CAP also changes the delivery guarantees for RFCOMM.

The primary concern is that changing the L2CAP retransmit policy may have much broader consequences than we would expect. If these consequences are acceptable (e.g., no other Bluetooth applications are running on the device, or they are able to deal with the reliability changes) then the retransmit policy can be quite useful.

ACL

The Asynchronous Connection-oriented Logical (ACL) transport protocol is one that you'll probably never use, but is worth mentioning because all L2CAP connections are encapsulated within ACL connections. Since RFCOMM connections are transported within L2CAP connections, they are also encapsulated within ACL connections. Two Bluetooth devices can have at most a single ACL connection between them, which is used to transport all L2CAP and RFCOMM traffic.

ACL is similar to IP in that it is a fundamental protocol that is rarely used to directly transport data. Instead, it is almost always used to encapsulate higher level protocol packets.

> *Note:* In the previous section, we said that L2CAP can be configured for varying levels of reliability. It is more accurate to say that the ACL connection between two devices can be configured for varying levels of reliability, which in turn affects all L2CAP and RFCOMM traffic between these devices. We prefer to say it about L2CAP because conceptually it doesn't make a difference, and then you don't have to think about ACL at all. Since ACL carries only L2CAP-related traffic, modifying the packet timeout for the ACL link is effectively the same as modifying the packet timeout for all L2CAP traffic.

SCO

The last transport protocol that we mention is the Synchronous Connection-Oriented (SCO) logical transport. This strange beast is a best-effort packet-based protocol that is exclusively used to transmit *voice-quality* audio – not just any audio, but voice-quality audio, at exactly 64 kb/s. It is useless for

transmitting CD-quality audio because the transmission rate isn't high enough, but it is just right for making phone calls. If you've used a Bluetooth headset, then your voice data is probably transmitted over an SCO connection. If you've used Bluetooth headphones to listen to your Bluetooth MP3 player, then the audio is probably transmitted over an L2CAP connection.

SCO packets are not reliable and never retransmitted, but there is a separate quality of service guarantee. An SCO connection is always guaranteed to have a 64 kb/s transmission rate. If other applications and connections on a device are contending for radio time to, say, transmit a file or synchronize a calendar, the SCO connection will be given priority. To ensure that all SCO connections have this guarantee, no Bluetooth device is allowed to have more than three active SCO connections. Furthermore, two Bluetooth devices can have at most one SCO connection between them. In practice, you'll seldom see any device with more than one active SCO connection at a time.

Applications transmitting data, regardless of whether they require reliably delivered packets, should almost always use L2CAP connections. SCO should only be used for transmitting voice-quality audio, and never for any other types of data, or even higher quality audio. In fact, the default SCO settings for most devices will not even transmit data packets transparently. Instead, the packets are encoded using the Continuously Variable Slope Delta audio codec. The received packets, once decoded, look nothing like the actual data transmitted, but *sound* the same. L2CAP and RFCOMM are the essential Bluetooth protocols and are discussed at length in subsequent chapters; SCO is of interest because of its widespread use with Bluetooth headsets.

Transport Protocol Summary

There are four major transport protocols defined by the Bluetooth specification. Of these, RFCOMM is often the best choice, and sometimes the only choice. L2CAP is also a widely used transport protocol that is used when the streaming nature of RFCOMM isn't needed.

Of the other two protocols, ACL is used to carry only L2CAP and RFCOMM-related traffic, and you probably will never end up using it directly. SCO is a highly specialized transport protocol designed specifically for voice-quality audio, and unless you're transmitting exactly that, you should probably stick with either RFCOMM or L2CAP.

Figure 1.3 illustrates the relationships between the major transport protocols. A connection is represented by a box, and a box within another box indicates that packets for the inner connection are encapsulated within packets for the outer connection. Two Bluetooth devices may have at most one SCO connection and one ACL connection between them, whereas the number of active L2CAP and RFCOMM connections are limited only by the number of available ports (discussed in the next section).

1.2.3 Port Numbers

A *port* is used to allow multiple applications on the same device to simultaneously utilize the same transport protocol. Almost all Internet transport protocols in common usage are designed with the notion of port numbers. Bluetooth is no exception, but uses slightly different terminology. In L2CAP, ports are called *Protocol Service Multiplexers* (PSM), and can take on odd-numbered values between 1 and 32,767. Don't ask why they have to be odd-numbered values, because you probably won't get a convincing answer. In RFCOMM, *channels* 1–30 are available for use. Throughout the rest of this book, the word *port* is used instead of protocol service multiplexer and channel, mostly for clarity.

Reserved/Well-Known Ports

Many transport protocols are designed with specific applications in mind, and at the very outset of designing the protocol, some of the ports are set aside and reserved for these applications. This set of ports is often referred to as the well-known, or reserved, ports for that protocol. It is expected that custom applications will not use any of the well-known ports, unless they are implementing a standardized service assigned to that port number. For example, in TCP/IP, port 80 is reserved for Web traffic, 23 is used for e-mail, and so on.

Bluetooth is no exception, and L2CAP reserves ports 1–1023 for standardized usage. For example, the Service Discovery Protocol (SDP; introduced next) uses port 1, and RFCOMM connections are multiplexed on L2CAP port 3. It turns out that RFCOMM does not have any reserved ports, which may not be that surprising given that it has so few port numbers in the first place. This information is summarized in Table 1.2.

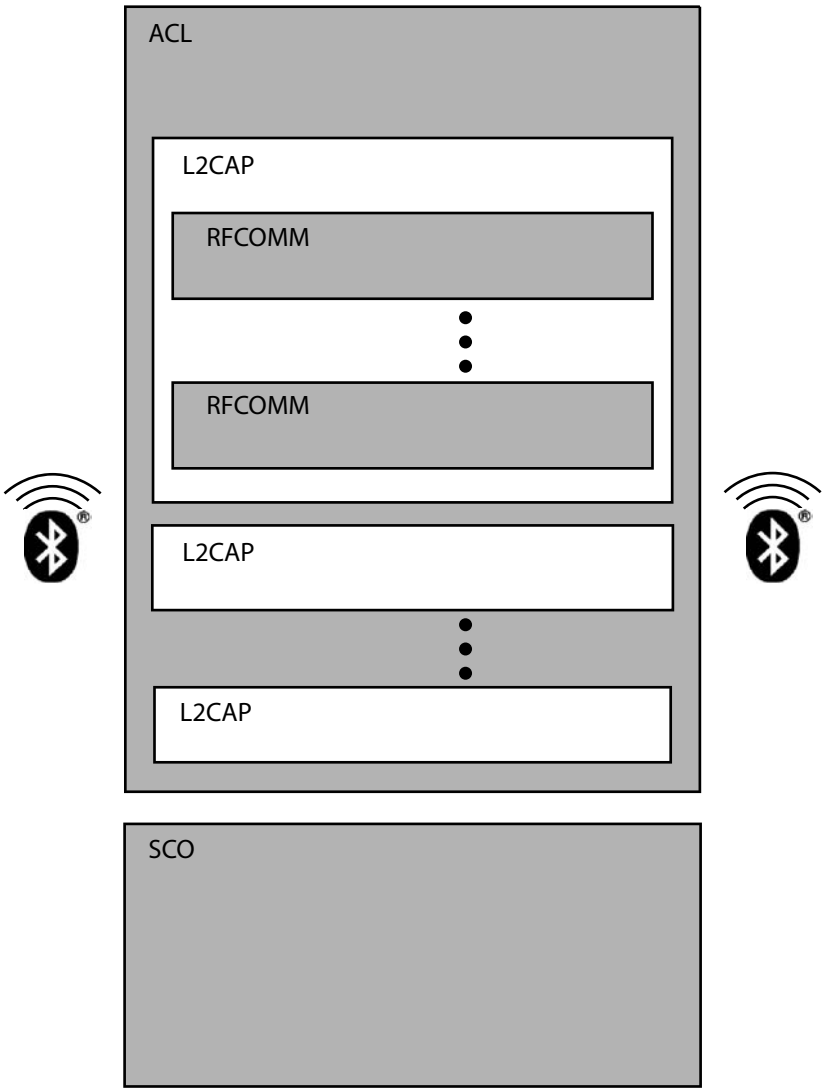


Figure 1.3 Some of the many Bluetooth Transport Protocols.

Table 1.2 Port numbers and their terminology for various protocols. Since two devices can have at most one SCO connection between them, there is no notion of ports in SCO.

Protocol	Terminology	Reserved/Well-Known Ports	Unreserved Ports
TCP	Port	1–1024	1025–65,535
UDP	Port	1–1024	1025–65,535
RFCOMM	Channel	None	1–30
L2CAP	PSM	Odd numbered 1–4095	Odd numbered 4097–32,765
SCO	N/A	N/A	N/A

1.2.4 Service Discovery Protocol

One of the big questions a client application must answer when establishing an outgoing connection is, *which port number is the server listening on?* If the server is a standardized application that uses one of the well-known port numbers, then the answer is easy. If not, then the traditional approach in Internet programming is to hard-code a port number in both the client and server applications. When it comes time to establish the connection, the server listens on that port, and the client connects to that port. The main disadvantage to this approach is that it is not possible to run two server applications which both use the same port number. Due to the relative youth of TCP/IP and the large number of available port numbers to choose from, this has not yet become a serious issue.

Bluetooth tries to avoid this problem by introducing the SDP. The basic premise is that every Bluetooth device maintains an SDP server listening on a well-known port number. When a server application starts up, it registers a description of itself and a port number with the SDP server on the local device. Then, when a remote client application first connects to the device, it provides a description of the service it's searching for to the SDP server, and the SDP server provides a listing of all services that match. Figure 1.4 illustrates this difference.

By using this approach, Bluetooth allows server applications to use *dynamically assigned* – when a server starts up, it can pick an arbitrary port number that is not being used on the local device, and thus ensure that it is free of

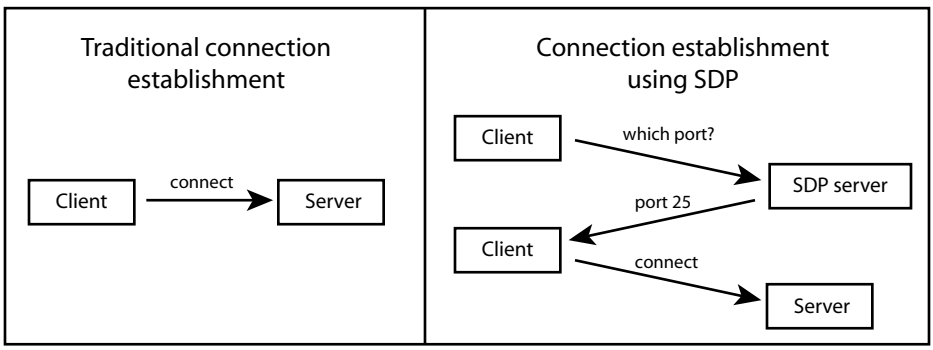


Figure 1.4 Traditional network applications use hard-coded port numbers, whereas most Bluetooth applications use an SDP server.

port conflicts. By decoupling the actual port used from the application itself, Bluetooth is able to avoid a few of the pitfalls that are a nuisance in Internet programming.

Service Record

The description of a service that a server application registers with its SDP server, and that the SDP server transmits to inquiring clients, is referred to as a *service record*, also an *SDP record*. In theory, the service record is quite simple; it consists of a list of *attribute/value* pairs. Each attribute is a 16-bit integer, and each value can be a basic data type, such as an integer or string, or a list of other data types. In some sense, a service record is essentially a dictionary of various entries that describe the service being offered. Figure 1.5 illustrates a service record and a few common attributes.

Next to the actual port number listed for a service, possibly the two most important attributes in a service record are the *Service ID* and the *Service Class ID List*. These two attributes are most often the ones used by a client to actually identify the desired service.

Service ID

Although having descriptions of services is nice, the big question that still needs to be answered is, how do clients know which service description is the one they seek? The approach taken in Bluetooth is to assign every single service a unique identifier, and make sure that the client already knows the

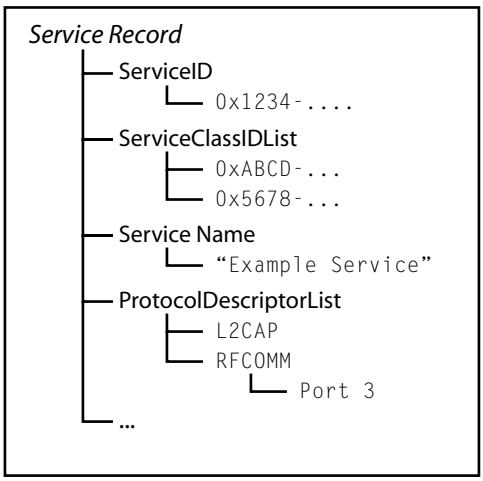


Figure 1.5 An SDP record is a list of attribute/value pairs, where each entry describes one aspect of the service offered.

unique ID of the service it's looking for. In other words, rather than assigning a port number at design time, developers assign a unique ID at design time. The big advantage that using a unique ID has over using port numbers is that the space of valid unique IDs is designed to be much larger than the space of valid port numbers, allowing developers to minimize the chance of unique ID conflicts.

This approach has been done before, and the Internet Engineering Task Force has a standard method for developers to independently come up with their own 128-bit Universally Unique Identifiers (UUID). This is the basic idea around which SDP revolves, and this identifier is called the service's *Service ID*. Specifically, a developer chooses this UUID at design time and when the program is run, it registers a service record containing its Service ID with the SDP server for that device. A client application trying to find a specific service would query the SDP server on each device it finds to see if the device offers any services with that same UUID.

The standard notation for a UUID is a hyphen-separated series of digits of the form "XXXXXXXX–XXXX–XXXX–XXXX–XXXXXXXXXXXX", where each "X" is a hexadecimal digit. The first segment of eight digits corresponds to bits 1–32 of the UUID, the next segment of four digits is bits 33–36, and so on.

Service Class ID List

Although a Service ID by itself can take us a pretty long way in terms of identifying services and finding the one we want, it's really meant for custom applications built by a single development team. The Bluetooth designers wanted to distinguish between these custom applications and classes of applications that all do the same thing. For example, two different companies might both release Bluetooth software that provides audio services over Bluetooth. Even though they're completely different programs written by different people, they both do the same thing. To handle this, Bluetooth introduces a second UUID, called the *Service Class ID*. Now, the two different programs can just advertise the same Service Class ID, and all will be well in Bluetooth Land. Of course, this is useful only if the two companies agree on which Service Class ID to use.

Another thought to consider is this: What if I have a single application that can provide multiple services? For example, many Bluetooth headsets can function as a simple headphone and speaker, and advertise that service class; but they are also capable of controlling a phone call – ansewring an incoming call, muting the microphone, hanging up, and so on. Although it's possible to just register two separate services in this case, each with a specific service class, the Bluetooth designers chose to allow every service to have a list of service classes that the service provides. So while a single service can have only *one* Service ID, it can have many *Service Class IDs*.

> *Note:* Technically, the Bluetooth specification demands that every SDP service record have a Service Class ID list with at least one entry. This can be an annoying bit to remember, but most Bluetooth implementations don't actually enforce this requirement.

Bluetooth Reserved UUIDs

Similar to the way L2CAP and TCP have reserved port numbers for special purposes, Bluetooth also has reserved UUIDs. Occasionally referred to as short UUIDs, these are mostly used for identifying predefined service classes, but also for transport protocols and profiles. (Bluetooth profiles are described in Section 1.3.6.) The lower 96 bits of all reserved Bluetooth UUIDs are the

Table 1.3 Example of reserved Bluetooth UUIDs. To obtain the full 128-bit UUID from a reserved UUID, shift it left by 96 bits and add the Bluetooth Base UUID.

SDP	0x0001
RFCOMM	0x0003
L2CAP	0x0100
SDP Server Service Class	0x1000
Serial Port Service Class	0x1101
Headset Service Class	0x1108

same, so they are typically referred to by their upper 16 or 32 bits. A few of the more common reserved Bluetooth UUIDs are shown in Table 1.3.

To get the full 128-bit UUID from a 16-bit or 32-bit number, take the *Bluetooth Base UUID* (`00000000-0000-1000-8000-00805F9B34FB`) and replace the leftmost segment with the 16-bit or 32-bit value. Mathematically, this is the same as

$$128\_bit\_UUID = 16\_or\_32\_bit\_number * 2^{96} + Bluetooth\_Base\_UUID$$

Common SDP Attributes

Bluetooth defines several reserved attribute IDs that always have a special meaning, and the rest can be used arbitrarily. Some of the more common reserved attributes are

Service Class ID List: A list of service class UUIDs that the service provides. This is the only mandatory attribute that must always appear in a service record.

Service ID: A single UUID identifying the specific service.

Service Name: A text string containing the name of the service.

Service Description: A text string describing the service provided.

Protocol Descriptor List: A list of protocols and port numbers used by the service.

Profile Descriptor List: A list of Bluetooth profile descriptor that the service complies with. Bluetooth Profiles are described in Section 1.3.6. Each descriptor consists of a UUID and a version number.

Service Record Handle: A single unsigned integer that uniquely identifies the record within the device.

SDP Record Structure – Detail

The basic building block of an SDP record is the *data element*. A data element is composed of a type, a size, and a value. The type of a data element can be either a basic type, such as integers and strings, or a sequence of data elements. The size indicates how much space the value takes up (in bytes), and the value is just the actual data to convey.

When we say that an SDP record is a list of attributes, we actually mean that every service record is a single data element with type sequence. The data elements within this sequence always alternate between 16-bit unsigned integers (the attribute) and some other type (the value) that varies with the attribute. For example, the type of the data element corresponding to the service name is always a string, but the service ID will be of type UUID.

1.2.5 Communicating via Sockets

Given the address of a device, the transport protocol, and the port, Bluetooth programming is essentially the tried and true network programmer's friend: sockets. A server application waiting for an incoming Bluetooth connection is conceptually the same as a server application waiting for an incoming Internet connection, and a client application attempting to establish an outbound connection behaves the same whether it is using RFCOMM, L2CAP, TCP, or UDP. For this reason, extending the socket programming framework to encompass Bluetooth is a natural approach. This section gives a brief introduction to the essential concepts behind socket programming without any distracting code.

Introducing the Socket

A *socket* in network programming represents the endpoint of a communication link. The idea is that from a software application's point of view, all data passing through the link must go into or come out of the socket. First used in the 4.2BSD operating system, sockets have since become the de facto standard for network programming.

There are two general ways to use a socket – as either a client or a server, (Figure 1.6). As before, the words client and server only refer to whether the socket is used to create outgoing connections, or accept incoming

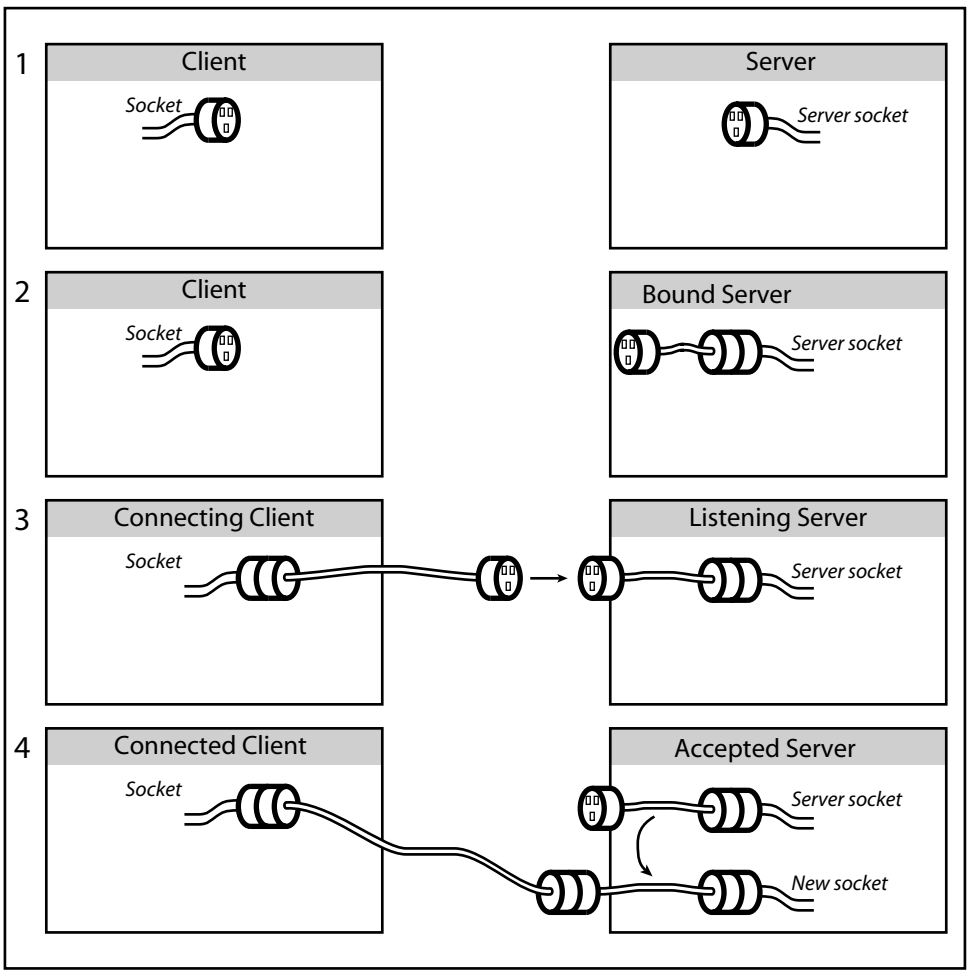


Figure 1.6 A conceptual illustration of the steps needed to obtain a connected socket. (1) When a socket is first created, it is mostly useless. (2) Binding a socket attaches it to a local Bluetooth adapter and port number. (3) When a server socket is placed in listening mode, then remote devices can initiate the connection procedure. (4) After a server socket has accepted a new connection, it spawns a new socket that is connected to the remote device.

connections. Just because a socket is used as a server socket doesn't actually mean that the application itself is a traditional "server" application.

The first step an application must take is to `create` a socket. Since sockets are used for many types of communication, the `create` command typically specifies the transport protocol to use. In Bluetooth programming, we'll almost always be creating either L2CAP or RFCOMM sockets.

The second step is different for client and server sockets:

Client Sockets

Client sockets are fairly simple in that once the socket has been created, only one additional step needs to be taken to obtain a connected socket. This is to issue a `connect` command, specifying the address and port of the target device. The operating system then takes care of all the lower level details, reserving resources on the local Bluetooth adapter, searching for the remote device, forming a piconet, and establishing a connection. Once the socket is connected, it can be used for data transfer.

Server/Listening Sockets

Server sockets are a bit more involved when it comes to obtaining a connected socket. Three steps must be taken: First, the application must `bind` the socket to local Bluetooth resources, specifying the local Bluetooth adapter and the port number to use.* Second, the `listen` command is used to place the socket into listening mode. This instructs the operating system to accept incoming connection requests on the local adapter and port number the socket was bound to. Finally, the `accept` command is used to obtain a connected socket that can be used for data transfer.

One of the major differences between a server socket and a client socket is that the server socket first created by the application can never be used for actual communication. Instead, each time the server socket accepts a new incoming connection using the `accept` command, it spawns a brand new socket that represents the newly established connection. The server socket then goes back to listening for more connection requests, and the application should use the newly created socket to communicate with the client.

Communicating Using a Connected Socket

Once a Bluetooth application has a connected socket, using it for communication is simple. The `send` and `receive` commands are used to . . . well, send and receive data. A send pushes out the bytes, but its successful return only means that the bytes have been moved from the application address space to the

* Most computers have only one Bluetooth adapter, so choosing a Bluetooth adapter isn't much of a choice at all.

operating system's buffers. They may or may not have actually left the device. A receive returns at least some data. It blocks until data, even 1 byte, are arrived. The receive will return with 0 bytes only when the connection is broken.

When the application is finished, it simply invokes the `close` command to disconnect the socket. Closing a listening server socket unbinds the port and stops accepting incoming connections.

Nonblocking Sockets with `select`

In the communications routines described so far, there is usually some sort of waiting involved. During this time, the controlling thread *blocks* (waits) and can't do anything else, such as respond to user input or operate on another socket. This may be fine (and may be even desirable) for simple applications, but is not acceptable for more complex programs that need to simultaneously perform other tasks. To avoid this pitfall of *synchronous* programming, it is possible to use multiple threads of control, with one thread dedicated to each task that requires some blocking. That can get quite hairy and complicated, though, so instead we'll turn to using *asynchronous* techniques as a solution.

The primary tool in asynchronous socket programming is a single *event loop*. The idea is that instead of waiting for a single event to happen (an incoming connection, or newly received data), the application waits for everything at once. If a single event of interest occurs, then the application processes that event and returns to the event loop where it waits for more events to occur.

In order for an event loop to work well, events must be processed quickly, without delay. Consequently, function calls should not block and must return immediately. To support this, sockets can be switched into *nonblocking* mode, so that all the operations that would normally block return immediately instead. When first created, a Bluetooth socket is by default a blocking socket. Table 1.4 describes the events that cause operations on blocking sockets to return. If not listed, then an operation generally returns immediately.

Since performing these operations on a nonblocking socket no longer informs the application of when these events occur, we need a different way to wait for them. The major programming environments discussed in subsequent chapters all support the `select` operation, which provides this

Table 1.4 Circumstances under which operations on a blocking socket return.

Operation	Returns When
`accept`	An incoming connection has been established
`connect`	Succesfully connected, or a timeout
`send`	The local operating system accepts responsibility for delivering the packet (not necessarily on succesful delivery)
`receive`	New data has arrived, or when disconnected

Table 1.5 Events detected by `select` for Bluetooth sockets. The `exception_set` is not used in Bluetooth.

Type of Socket	`read_set`	`write_set`
Connected socket	Incoming data ready or connection terminated	Remote device can accept data
Listening socket	Incoming connection established	Never
Connecting socket	Never	Connection succeeded or completely failed

functionality. We'll see it in different forms, but a call to select typically takes four parameters:

```
select( read_set, write_set, exception_set, timeout )
```

Each of the first three parameters contains a set of sockets. The `select` operation waits for one of these events to occur or for the timeout period to expire. Table 1.5 describes the events detected by `select` when a socket is in a particular set. Notice that the third set, `exception_set`, does not correspond to any events for any types of sockets. Although it has meaning in other types of sockets, such as TCP sockets, this set is not used in the major `select` implementations for Bluetooth sockets.

If `select` could only be used to detect events for Bluetooth sockets, it still wouldn't be all that useful. However, most implementations can fuse event detection for many different I/O modalities, including TCP sockets, file handles, and user input handles. Seen this way, applications that block largely on I/O operations are well served with a `select`-based event loop.

Table 1.6 The three Bluetooth power classes.

Power Class	Transmission Power Level (mW)	Advertised Range (m)
1	100	100
2	2.5	10
3	1	1

1.3 Useful Things to Know about Bluetooth

This section covers concepts that are useful to know about Bluetooth, but may not have a direct translation into Bluetooth programming methods or techniques. Understanding the material in this section is not necessary to get started with Bluetooth programming; however, it might be helpful for designing an effective and efficient Bluetooth application. Readers eager to get started writing Bluetooth applications can skip to the next chapter without fear.

1.3.1 Communications Range

Bluetooth devices are divided into three power classes; the only difference between them is the transmission power levels used. Table 1.6 summarizes their differences. Almost all Bluetooth-enabled cell phones, headsets, laptops, and other consumer-level Bluetooth devices are class 2 devices. Class 1 devices are also widely available to consumers, but class 3 Bluetooth devices are rare at the time of this writing and we suspect they will remain so.

The ranges given in Table 1.6 are only rough estimates used for advertising purposes. In practice, the useful range of a Bluetooth device can vary significantly with the environment and conditions of use. Walls, buildings, trees, desks, and other obstructions all affect the strength of a signal. Other wireless technologies being used in close proximity such as 802.11 and some cordless phones may also affect Bluetooth. Microwave ovens in particular can sometimes render Bluetooth devices unusable (see Section 1.3.3). Additionally, people are actually quite good at blocking Bluetooth signals, mostly because water (which constitutes around 60% of the human body) does a great job in absorbing radio waves at the frequencies used by Bluetooth.

1.3.2 Communications Speed

It is also difficult to give a reliable estimate on the bandwidth of a Bluetooth communications channel, but ballpark figures do help. Theoretically, two Bluetooth devices have a maximum assymetric data rate of 723.2 kilobits per second (kb/s) and a maximum symmetric data rate of 433.9 kb/s. Here, asymmetric means that only one Bluetooth device is transmitting, and symmetric means that both are transmitting to each other. In practice, the transfer rates are likely to be less, since there's always going to be noise on wireless communications channels, as well as some transport protocol overhead on each packet transmitted.

Like all wireless communications methods, the strength of a Bluetooth signal deteriorates quadratically with the distance from the source. Since weaker signals are much more likely to be corrupted by noise, the maximum communication speed between two Bluetooth devices is strongly limited by their separation distance. Unless the distance and obstructions between two Bluetooth devices can be closely controlled, it's a good idea to design a protocol that can tolerate lower communication speeds or dropped packets.

Bluetooth devices that conform to the Bluetooth 2.0 specification, released in late 2004, have a theoretical limit triple that of older devices (2178.1 kb/s asymmetric and 1306.9 kb/s symmetric), and at the time of this writing (August, 2006), are beginning to gain market share. These are mostly backward compatible with older Bluetooth devices, but are limited to the older speeds when communicating with them.

1.3.3 Radio Frequencies and Channel Hopping

All Bluetooth devices operate in the 2.4-GHz frequency band. This means that they use the same radio frequencies as microwaves, 802.11, and some cordless phones (the kind that attach to landlines, not mobile phones). What makes Bluetooth different from the other technologies is that it divides the 2.4-GHz band into 79 channels and employs channel-hopping techniques so that Bluetooth devices are always changing which frequencies they're transmitting and receiving on. Thus, interference on one channel does not matter very much, since there should not be any interference after a hop to a new channel.

For comparison, take a look at the way wifi (i.e., 802.11b and 802.11g) operates. The 2.4-GHz band is divided into 14 channels that are 5 MHz

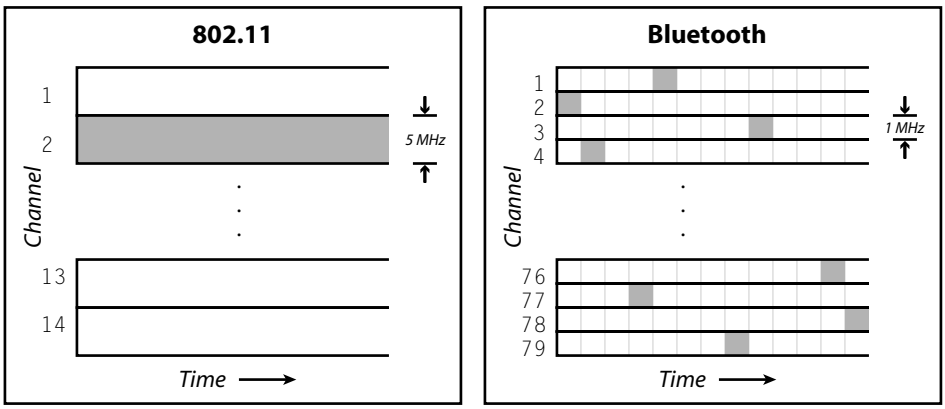


Figure 1.7 Both 802.11 and Bluetooth use the 2.45-GHz band for communication, but Bluetooth divides the band into 79 channels, whereas 802.11 divides the band into 14 channels. Another major difference is that a single 802.11 network typically picks a channel and stays with it, whereas devices on a Bluetooth piconet change radio frequencies very rapidly.

wide. When a wireless network is set up, the network administrator chooses one of these channels and all 802.11 devices on that wireless network will always transmit on the radio frequency for that channel: (Sometimes this is done automatically by the wireless access point.) If there are many wireless networks in the same area, like in an apartment building where every apartment has its own wireless router, then chances are that some of these networks will collide with each other and their overall performance will suffer.

Bluetooth, like 802.11, divides the 2.4-GHz band into channels, but that's where the similarity ends. For starters, it has 79 channels instead of 14, and the channels are narrower (1 MHz wide instead of 5 MHz). The big difference, though, is that Bluetooth devices never stay on the same channel. An actively communicating Bluetooth device changes channels every 625 μs (1600 times per second). It tries to do this in a random order so that no one channel is used much more than any other channel. Of course, two Bluetooth devices that are communicating with each other must hop channels together so that they're always transmitting and receiving on the same frequencies. This is illustrated in Figure 1.7.

Supposedly, all this hopping around makes Bluetooth more robust to interference from nearby sources of evil radio waves, and allows for many Bluetooth networks to coexist in the same place. Newer versions of Bluetooth

(1.2 and greater) go even further and use *adaptive frequency hopping*, where devices will specifically avoid channels that are noisy and have high interference (e.g., a channel that coincides with a nearby 802.11 network). How much it actually helps is debatable, but it certainly makes Bluetooth a lot more complicated than the other wireless networking technologies.

1.3.4 Bluetooth Networks – Piconets, Scatternets, Masters, and Slaves

To support the intricacies of a pseudorandom channel-hopping scheme, Bluetooth uses some even more confusing terminology that doesn't matter all that much when developing Bluetooth software. Since it's mentioned in a lot of Bluetooth literature, we'll describe it here, but don't put too much effort into remembering it.

Two or more Bluetooth devices that are communicating with each other and using the same channel-hopping configuration (so that they're always using the same frequencies) form a Bluetooth *piconet*. A piconet can have up to eight devices in total. That's pretty straightforward. But how do they all agree on which frequencies to use and when to use them? That's where the *master* comes in. One device on every piconet is designated the master, and has two roles. The first is to tell the other devices (the *slaves*) which frequencies to use – the slaves all agree on the frequencies dictated by the master. The second is to make sure that the devices communicate in an orderly fashion by taking turns.

To better understand the master device's second role, consider wifi (802.11) where there is no such thing as an orderly way of transmitting. A device wishing to send a data packet first listens for silence (waits until no other device is transmitting), and then waits a little more, before finally transmitting its data. The recipient replies with an acknowledgment. If the sender doesn't get the acknowledgment, then it tries again. You can see how this can get messy when a lot of 802.11 devices are trying to transmit at the same time. The lack of an acknowledgment can mean either the data did not arrive or it did but the acknowledgment failed. Bluetooth, on the other hand, uses a turn-based transmission scheme, where the master of a piconet essentially informs every device when to transmit and when to keep quiet. The big advantage here is that the data transfer rates on a Bluetooth piconet will be somewhat predictable, since every device will always have its turn to transmit. It's like the difference between a raucous town meeting where

everyone is shouting and a moderated discussion where the moderator gives everyone who raises a hand a chance to speak.

The last bit of Bluetooth networking terminology here is the *scatternet*. It's theoretically possible for a single Bluetooth device to participate in more than one piconet. In practice, a lot of devices don't support this ability, but it is possible. When this happens, the two different piconets are collectively called a scatternet. Despite the impressive name, don't get too excited because scatternets don't really do a whole lot. In fact, they don't do anything at all. In order for two devices to communicate, they must be a part of the same piconet. Being part of the same scatternet doesn't help, and the device that joins the two piconets (by participating in both of them) doesn't have any special routing capabilities. Scatternet is just a name, and nothing more.

To be clear, the reason why all this talk about piconets, scatternets, masters, and slaves doesn't matter is that for the most part, all of this network formation and master–slave role selection is handled automatically by Bluetooth hardware and low-level device drivers. As software developers, all we need to care about is setting up a connection between two Bluetooth devices, and the piconet issue is taken care of for us. But it does help to know what the terms mean.

1.3.5 Security – PINs and Link Keys

Two Bluetooth devices can conduct an authentication procedure, where they verify their respective identities. Once authenticated, they have the option of encrypting all data packets they exchange. Authentication can also be performed without encryption, and is useful when the two devices only need to be confident that they're communicating with the right machine, but don't actually care about eavesdroppers.

The principle upon which all this relies is the *shared secret*. This is more or less the same as the secret password that you may have shared with your best friend as a child, or the secret handshake used by secret societies all around the world; because the two devices share a secret that no other devices have, they can leverage that secret to establish a secure communications channel. The shared secret in Bluetooth is known as the *PIN* – an alphanumeric sequence up to 16 characters long.

The first time two devices go through authentication or encryption is known as the *pairing* procedure. First, a common PIN must be given to the

devices, either by hard-coding it in or by prompting the user to enter it in via keypad or other input mechanism. Next, each device uses the PIN to generate a *link key*. The link key is saved on both devices, and is what's actually used to encrypt data transmitted between the two. Note that during the pairing process, the PIN is never transmitted over the air.

Once two devices have been paired, applications on each device can request authentication or encryption as needed. Encryption takes effect at a low level (the physical link level) and so if packets for one connection are encrypted between two devices, then packets for all connections between them are encrypted. Because of this, authentication and encryption are typically handled by the operating system directly, instead of by applications.

An important consideration to keep in mind about authentication and encryption is that neither procedure is foolproof, and they should be thought of as providing light to moderate security. That is to say, don't rely on the default Bluetooth security model if your life depends on the secrecy of the data your Bluetooth device is transmitting. But otherwise, it's strong enough to stop a nosy passerby.

Security Mode

Occasionally, you may see the term *security mode* thrown around. Like the term scatternet, it's not terribly useful, but is worth knowing about. Briefly put, the security mode of a Bluetooth device determines its policy on if and when to demand encryption from connected devices.

Security Mode 1: The device never initiates authentication or encryption with a connected device. However, it must comply if the connected device requests either.

Security Mode 2: The device initiates authentication and encryption when requested by individual local applications. All major operating systems operate in this mode by default.

Security Mode 3: In this infrequently used security mode, the device initiates authentication and encryption as soon as any connection is established, and refuses to communicate with unpaired devices.

Simple Pairing

A significant problem with the PIN-exchange method in earlier versions of Bluetooth is that users often choose a lousy PIN, such as 1234 or 0000.

A second problem is that devices without input capabilities, such as most headsets, typically have a trivial PIN (almost always 0000) hard-coded to the device, which defeats the purpose of the PIN. Furthermore, requiring users to actually choose and input a PIN often resulted in clunky and confusing user interfaces.

To address these problems, Bluetooth 2.1 introduces the *Simple Pairing* procedures. The most significant change is that in most pairing situations, the user will not need to enter a PIN. Instead, the PIN is automatically generated, and the user is only prompted to accept or reject a pairing. Additionally, Simple Pairing introduces stronger encryption techniques that are already commonly used by standard Internet protocols such as SSH, IPSec, PGP, and SSL. These two enhancements result in a much stronger security model that is ultimately more convenient for the end user.

1.3.6 Bluetooth Profiles + RFCs

Bluetooth programming defines transport protocols and methods of communicating, but it also goes one step further to specify methods of using Bluetooth to accomplish higher level tasks. These methods and specifications are collectively called the *Bluetooth Profiles*, and define standardized ways to perform tasks such as transferring files, playing music, using nearby printers, and so on.

Internet programmers may notice a resemblance with the Request for Comments (RFC) specifications managed by the Internet Engineering Task Force, which detail the standard ways of sending e-mail, browsing the World Wide Web, routing data packets, and all the common applications that come to mind when the term Internet is invoked. Bluetooth Profiles are identical in nature to the Internet RFCs, but also complementary. Due to the short-range nature of Bluetooth, the Bluetooth Profiles emphasize tasks that can assume physical proximity.

Some of the more well-known and widely used Bluetooth Profiles include

OBEX Object Push: Allows devices to send and receive arbitrary data files, which are often documents, images, sounds, or business cards (vCards). OBEX stands for "object exchange," which can be either pushed or pulled.

File Transfer: Allows one device to access the filesystem of another device to send and receive files. This is different from OBEX Object Push in

that OBEX Object Push is like saying "Here, take this file," and File Transfer is like saying "Let me look at all of your files and do stuff (upload/download/rename/copy/etc.)"

Dial-Up Networking: Allows devices to use cellular phones (or other Bluetooth devices connected to a phone line) as modems to place phone calls and connect to the Internet.

Hands-Free Audio: Allows Bluetooth headsets to connect to other Bluetooth devices such as cellular phones or computers and transmit voice-quality audio. This profile uses SCO connections.

Advanced Audio Distribution: Allows Bluetooth headsets to connect to other Bluetooth devices and transmit high-quality audio, such as MP3s and other music. This profile uses L2CAP connections.

Personal Area Network: Allows Bluetooth devices to form IP networks, and especially to share one device's Internet connection with the other.

Human Interface Device: Allows Bluetooth mice, keyboards, and other input devices to be used with Bluetooth-enabled computers.

Serial Port Profile: Allows RFCOMM connections between two Bluetooth devices to be treated as serial cable connections.

This is certainly not a comprehensive list of all Bluetooth Profiles, just some of the more popular ones. If you find yourself needing to implement or understand one of the Bluetooth Profiles, you can find the specification and all the details for that particular profile on the Bluetooth Web site,* where they are freely distributed.

1.3.7 Host Controller Interface

Up to this point, we've glossed over most of the low-level details concerning how a Bluetooth adapter is actually controlled and used. The Host Controller Interface (HCI) defines how a computer (the *Host*) interacts and communicates with a local Bluetooth adapter (the *Controller*). All communications that occur between the two are encapsulated within HCI packets. Four types of packets are defined:

Command packet: This type of packet is sent from the host computer to the Bluetooth adapter, and is used to control the adapter. Packets of

* http://www.bluetooth.com/Bluetooth/Learn/Technology/Specifications/

this type can be used to start a device inquiry, connect to a remote device, adjust connection parameters, and so on.

Event packet: This type of packet is generated by the Bluetooth adapter and sent to the host computer whenever an event of interest occurs. Common events include a detected device, a connection establishment, information about the local Bluetooth adapter, and so on.

ACL data packet: HCI packets of this type encapsulate data destined for or received by a remote Bluetooth device. In this sense, HCI is a transport protocol for all ACL, L2CAP, RFCOMM, and higher level transport protocols. Once the packet passes through the Bluetooth adapter, however, the HCI headers are stripped off and the bare ACL packet is transmitted over the air.

Synchronous data packet: SCO data packets are also encapsulated within HCI packets when first transmitted from the host to the Bluetooth adapter, and are wrapped within this type of packet. As with ACL, the HCI headers are stripped from the packet when it's actually transmitted over the radio, and are rewrapped after being received.

To perform low-level Bluetooth operations, an application typically requires access to the HCI layer of Bluetooth communications. Many programming environments do not expose this layer, and instead provide higher level wrappers that are both easier to use, but also more restrictive. In general, there is no need to directly use the HCI protocol, and will instead use these wrappers.

An important thing to remember about HCI is that it applies only to a computer and its local Bluetooth adapters. It is not possible for two separate Bluetooth devices to have an HCI connection between them, nor is it possible for one host computer to have an HCI connection to a remote Bluetooth device that is not physically attached to the host.

1.3.8 Limitations – Things Bluetooth Can't Do

Although Bluetooth is one of many wireless communications technologies, it does not have the same characteristics as all the others. Here, we list a few capabilities sometimes present in other specifications that are either not present or highly limited in Bluetooth.

Announce the presence of a device: It is not possible for a Bluetooth device to do the equivalent of shouting, "Hello! Here I am!" It is only possible to inquire for nearby devices – similar to saying "Hello? Who's there?"

Detect when a remote device is inquiring for nearby devices: Although Bluetooth exposes many low-level events, it does not expose an event that triggers when an inquiry message is detected. The Bluetooth adapter can detect this event, of course, but it just doesn't pass that information on to the host computer. It is possible to build a custom hardware solution to detect this, but this option is most likely not feasible for most software developers.

Determine the Bluetooth address of an inquiring device: A Bluetooth device conducting a device inquiry never transmits information identifying itself. Thus, even if it were possible to detect in software when a remote device is conducting an inquiry, it still wouldn't be possible to identify that device.

Distance to a remote Bluetooth device: Many programmers poking around in the Bluetooth specification discover that some devices report the strength of a signal for a newly discovered or already connected device. It is tempting to try and use this information to infer how far one is from that device, but ultimately frustrating. The radio continually adjusts the signal strength both to reduce transmission errors and to conserve power. Interference from other radio sources, multipath effects, and signal attenuation through different materials all serve to confound the signal strength measurements.

Broadcast messages: It is not possible for a Bluetooth device to broadcast a message to all nearby devices. There is a protocol for the master of a piconet to broadcast a message to all its slaves using connectionless L2CAP packets, but it is rarely useful due to the requirement that all devices be in the same piconet and also because the protocol is not reliable. (Broadcast messages are not acknowledged or retransmitted.)

1.3.9 Device Discovery Explained

Bluetooth splits the 2.4-GHz band into 79 channels, with the intent that all devices on a single piconet use exactly one of these channels at a time. Of these 79 channels, 32 are also used for detecting nearby devices and establishing connections. An inquiring device transmits inquiry messages on

these channels, and a discoverable Bluetooth device periodically listens for and replies to the inquiry messages.

Inquiring for Nearby Devices

To conduct a device inquiry, the inquiring device divides the 32 channels into two sets, called train A and train B. These sets are more or less randomly chosen, and a discoverable device is no more likely to be listening on a channel in train A than it is on a channel in train B.

Next, the inquiring device transmits an inquiry message on each channel in train A, listening for a response each time it changes channels. If there is no reply, this is repeated at least 256 times (more if the inquiring device has active SCO connections). Once train A has been exhausted, the inquiring device repeats the process for train B, alternately transmitting inquiry messages and listening for responses for each channel in train B.

A single transmit/listen cycle for each channel takes 625 μs. Since there are 16 channels in each train, and each train is repeated 256 times, the inquiring device spends $625\,\mu\text{s} \times 16 \times 256 = 2.56\,\text{s}$ in each train before switching. By default, a single device inquiry switches trains four times for a total device inquiry lasting $2.56 \times 4 = 10.24\,\text{s}$.

Listening for Incoming Connections/Inquiries

A discoverable Bluetooth device periodically enters the inquiry scan substate. This is slightly different from the inquiry scan option mentioned in Section 1.2.1; the inquiry scan option controls whether or not a device ever enters the inquiry scan substate.

Two parameters, the inquiry scan interval and the inquiry scan window, determine how frequently a device enters the inquiry scan substate and how much time the device spends there, respectively. By default, the scan interval is 2.56 s, and the scan window is 11.25 ms. Both of these parameters can be adjusted using HCI commands, although it's generally not a good idea to do so.

Irrespective of the scan interval and the scan window, the channel on which a discoverable device listens for inquiry messages changes every 1.28 s. Each time the channel changes, the next one is chosen pseudorandomly, without trying to predict where an inquiring device is likely to transmit.

Looking at the inquiry and inquiry scan processes side by side, it should be evident that detecting a nearby device is not an instantaneous process. For example, as long as the channel a device is listening on is not in the train that an inquiring device is transmitting on, the two shall never meet. This game of whack-a-mole often continues for many seconds even when the two devices are within inches of each other.

In an effort to alleviate this problem, Bluetooth 1.2 and more recent devices may also enter the *interlaced inquiry scan* substate instead of the standard inquiry scan substate. In the interlaced mode, each time the device enters the scan window, it listens on two different channels instead of just one. The intent is to maximize the chance that an inquiring device actually transmits on the channel that matches the discoverable device.

1.3.10 Bluetooth Stacks

Throughout the rest of this book, we will frequently use the term *stack* to refer to a Bluetooth software environment. A stack simply refers to a collection of device drivers, development libraries, and user-level tools provided by an organization that allows software developers to create Bluetooth applications, and allows users to use the Bluetooth capabilities of a device.

In most popular operating systems, there is typically one dominant Bluetooth stack. For example, GNU/Linux, OS X, Symbian OS, and Palm OS all have a single stack that almost all programmers targeting those platforms use. Since applications written for one stack are not compatible with other stacks, having a single dominant stack usually provides a more consistent and simpler experience for the end user.

The major exceptions to this rule are Microsoft Windows XP and Pocket PC, where both platforms have two competing Bluetooth stacks each. This is discussed more in Chapter 4, but the simple way of putting it is to say that creating effective Bluetooth applications for Microsoft devices can be a bigger headache than for other platforms.

As we discuss each stack in detail, we'll be most concerned with the development libraries provided and the functionality exposed to software applications. Since Bluetooth is a communications technology, the transport protocols supported by each stack are of particular interest. All of the stacks introduced in this book and the basic transport protocols supported by each are listed in Table 1.7.

Table 1.7 Bluetooth stacks and wrappers, and the transport protocols supported, listed in the order introduced in this book. RFCOMM is common to all environments, which may be a factor when designing applications. The Widcomm and Series 60 C++ interfaces are not introduced in this book, but are included here for comparison.

PyBlueZ (GNU/Linux)	RFCOMM	L2CAP	SCO	HCI
PyBlueZ (Windows XP)	RFCOMM			
BlueZ (GNU/Linux)	RFCOMM	L2CAP	SCO	HCI
Microsoft (Windows XP)	RFCOMM			
Widcomm (Windows XP)	RFCOMM	L2CAP	SCO	HCI
Java – JSR82 (cross-platform)	RFCOMM	L2CAP		
Series 60 Python (Symbian OS)	RFCOMM			
Series 60 C++ (Symbian OS)	RFCOMM	L2CAP		HCI
OS X	RFCOMM	L2CAP		HCI

Occasionally, wrappers around the development libraries for a stack are created to provide a Bluetooth programming interface in a higher level programming language, such as Python or Java. These are sometimes referred to as stacks themselves and we will compare them with stacks, but are more accurately named wrappers.

1.4 Summary

There are only a few essential concepts needed to get started with Bluetooth programming and only a few important considerations to keep in mind when designing a Bluetooth application. Bluetooth programming is closely related

to Internet programming; techniques for one easily translate into techniques for the other, with the primary difference being that Bluetooth devices can assume physical proximity. The techniques introduced in this chapter can be applied to all major operating systems with little or no adaptation. The following chapters show exactly how to do this for a variety of platforms and programming languages.

Although we have introduced the parts of Bluetooth most relevant to a software developer, by no means have we provided a comprehensive discussion of Bluetooth. As previously indicated, we have deliberately left out significant parts of the Bluetooth specification in favor of brevity and simplicity.

A detailed introduction to the Bluetooth Profiles is beyond the scope of this book, and is easily an entire book (or number of books) by itself. However, it is our belief that understanding the material presented in this chapter will make it significantly easier and faster to understand any part of the Bluetooth specification or any Bluetooth Profile of which the reader requires in-depth knowledge.

Bluetooth programming with Python

Device discovery	`discover_devices()`
Name lookup	`lookup_name( address )`
SDP connect SDP search	`find_service( uuid )`
Establish an outgoing connection	`s = BluetoothSocket( protocol )` `s.connect( ( address, port ) )`

Establish an incoming connection	`s = BluetoothSocket( protocol )`
	`s.bind( ( address, port ) )`
	`s.listen( backlog )`
	`s.accept()`

Advertise an SDP service	`advertise_service( s, name, ... )`
Transfer data	`s.send( data )` `s.recv( numbytes )`
Disconnect	`s.close()`

PyBluez Quick Reference

Chapter 2

Bluetooth Programming with Python

It's time to get our hands dirty and learn how to implement the Bluetooth essentials, using the Python programming language. Why Python, you ask? Why not start with a very widely used language such as Java, C (or insert your favorite language here)? There are two answers to this question. The short answer is that it's just plain easy, and the code is short and readable. The long answer is that Python is a versatile and powerful dynamically typed object-oriented language, providing syntactic clarity along with built-in memory management so that the programmer can focus on the algorithm at hand without worrying about memory leaks or matching braces. Additionally, there's no need to worry about compiling object files or linking against libraries or setting the correct classpaths because, for our purposes, Python "Just Works," and the reader can quickly see the essential concepts in action.

If you're not very comfortable with Python, don't worry. The examples use only the simplest parts of Python and can be read as pseudocode. But if you want to run the examples, you might need some help getting started. Although Python has a large and comprehensive library, Bluetooth is not yet part of the standard distribution. Enter *PyBluez*, a Python extension that provides access to system Bluetooth resources on Windows XP and GNU/Linux computers. Once installed, as described in Section 2.7, we're ready to get up and running.

Table 2.1 Transport protocols supported by PyBluez.

PyBluez (GNU/Linux)	RFCOMM	L2CAP	SCO	HCI
PyBluez (MS Windows)	RFCOMM	L2CAP	SCO	HCI

We believe this chapter will make it easier to follow the subsequent chapters that address other languages. The data structures in other programming languages can be very involved and tend to obscure the basic operations. Python does not suffer from this drawback, and serves as an excellent learning medium.

Table 2.1 illustrates the transport protocols supported by the GNU/Linux and MS Windows implementations of PyBluez. The Windows XP version of PyBluez is somewhat less functional in Microsoft Windows than is the GNU/Linux version. The differences will be marked in the section headers throughout this chapter. In particular, it does not expose an API for the L2CAP and HCI layers of Bluetooth (see Chapter 4 for more information). It requires the *Microsoft Bluetooth stack*, and will not work with other Bluetooth stacks, such as the one provided by the Widcomm environment.

Bluetooth communication is all about interacting with devices that are physically close to each other. It is natural to begin by finding nearby devices, and the chapter begins with a very short Python program to detect and display nearby Bluetooth devices. This section is followed by examples and explainations as to how to communicate using RFCOMM and L2CAP protocols. The third section delves into the details, discovering the services provided by these nearby devices. One may know the address of a device, its user-friendly name, or simply find a device that provides a needed service. The process of advertising and searching for a service is quite involved and so it comes after the very basic functions describe in the first two sections. The later half of the chapter can be considered more advanced. Several topics are addressed, including asynchronous and audio communications.

2.1 Choosing a Communication Target

We begin by showing how to choose a communication target. Example 2.1 looks for a nearby device that has the user-friendly name “My Phone.”

Example 2.1 findmyphone.py

```
from bluetooth import *

target_name = "My Phone"
target_address = None

nearby_devices = discover_devices()

for address in nearby_devices:
    if target_name == lookup_name( address ):
        target_address = address
        break

if target_address is not None:
    print "found target bluetooth device with address", target_address
else:
    print "could not find target bluetooth device nearby"
```

A Bluetooth device is uniquely identified by its address, so choosing a communication target amounts to picking a Bluetooth address. If only the user-friendly name of the target device is known, then two steps must be taken to find the correct address. First, the program must scan for nearby Bluetooth devices. The function `discover_devices` does this and returns a list of addresses of detected devices. Next, the program uses `lookup_name` to connect to each detected device, request its user-friendly name, and compare the result to the desired name. In this example, we just assumed that the user is always looking for the Bluetooth device named "My Phone," but we could also display the names of all the Bluetooth devices and prompt the user to choose one. Pretty easy, right?

Bluetooth Addresses in Python

PyBluez represents a Bluetooth address as a string of the form "xx:xx:xx:xx:xx:xx", where each x is a hexadecimal character representing 4 bits of the 48-bit address, with the most significant bits listed first. Bluetooth devices in PyBluez will always be identified using an address string of this form. The output resulting from executing the code in Example 2.1 might have the following output if the target device had address "01:23:45:67:89:AB":

```
# python findmyphone.py
found target bluetooth device with address 01:23:45:67:89:AB
```

Device Discovery and Name Lookup

The function `discover_devices` is used in Example 2.1 without any arguments, which should be sufficient for most situations, but there are two optional parameters that permit finer control over the inquiry process:

```
discover_devices( duration=8, flush_cache=True )
```

When a Bluetooth device is detected during a scan, its address is cached for up to a few minutes. By default, `discover_devices` will return addresses from this cache in addition to devices that were actually detected in the current scan. To ensure that the function returns only the addresses of those devices that respond to the inquiry performed during this call, set the `flush_cache` parameter to `True`.

The amount of time that `discover_devices` spends scanning for devices can be controlled via the parameter `duration`, which is specified in integer units of 1.28 s. This somewhat strange number is a consequence of the Bluetooth specification: a device inquiry always lasts for a multiple of *exactly* 1.28 s. It's usually not a good idea to decrease this below the default value of 8 (10.24 s).

The function `lookup_name` also takes an optional parameter that controls how long it spends searching:

```
lookup_name( address, timeout=10 )
```

If `lookup_name` is not able to determine the user-friendly name of the specified Bluetooth device within a default value of 10 s, then it gives up and returns `None`. Setting the `timeout` parameter, a floating point number specified in seconds, adjusts this timeout.

An important property of Bluetooth is that wireless communication is never perfect, so `discover_devices()` will sometimes fail to detect devices that are in range, and `lookup_name()` will sometimes return `None` when it shouldn't. Unfortunately, it's impossible for the program to know whether these failures were a result of a bad signal or if the remote devices really aren't there any more. In these cases, it may be a good idea to try a few times, or to adjust the search durations.

2.2 Communicating with RFCOMM

The basics of how to establish a connection using an RFCOMM socket, transfer some data, and disconnect are exemplifed in Examples 2.2 and 2.3. In the first example, a server application waits for and accepts a single connection on RFCOMM port 2, receives some data, and prints it on the screen. In the second example, the client program connects to the server, sends a short message, and then disconnects.

Example 2.2 rfcomm-server.py

```
from bluetooth import *

port = 1
backlog = 1

server_sock=BluetoothSocket( RFCOMM )
server_sock.bind(("",port))
server_sock.listen(backlog)

client_sock, client_info = server_sock.accept()
print "Accepted connection from ", client_info

data = client_sock.recv(1024)
print "received:", data

client_sock.close()
server_sock.close()
```

Example 2.3 rfcomm-client.py

```
from bluetooth import *

server_address = "01:23:45:67:89:AB"
port = 1

sock = BluetoothSocket( RFCOMM )
sock.connect((server_address, port))

sock.send("hello!!")

sock.close()
```

In the socket programming model, a socket represents an end point of a communication channel. Sockets are not connected when they are first created, and are useless until a call to either `connect` (client application) or `accept` (server application) completes successfully. Once a socket is connected, it can be used to send and receive data until the connection fails due to link error or user termination.

An instance of the `BluetoothSocket` class represents a single Bluetooth socket in PyBluez, and almost all communications will use methods of this class:

```
sock = BluetoothSocket( protocol )
```

The constructor takes in only one parameter specifying the type of socket. This can be either `RFCOMM`, as used in these examples, or `L2CAP`, which is described in the next section. The construction of the socket is the same for both client and server sockets.

Incoming Connections

An RFCOMM `BluetoothSocket` used to accept incoming connections must be attached to operating system resources with the `bind` method:

```
sock.bind( address_and_port )
```

The method `bind` requires a single parameter – a tuple specifying the address of the local Bluetooth adapter to use and a port number. Usually, there is only one local Bluetooth adapter or it doesn't matter which one to use, so the empty string indicates that any local Bluetooth adapter is acceptable. Once a socket is bound, a call to the method `listen` puts the socket into listening mode to make it ready to accept incoming connections with the `accept` method:

```
sock.listen( backlog )
```

The method `listen` requires a single parameter, `backlog`, which will almost always be set to 1. In between the time a program calls `listen` and `accept` (or between two calls to `accept`), the operating system may accept an incoming connection and hold it in a pending state until the server application calls `accept`. This parameter controls how many of these pending incoming connections the operating system should accept before it begins rejecting incoming connection requests. It is more useful in Internet programming,

where a busy server may accept hundreds or thousands of connections each second. In the context of Bluetooth, where only a handful of devices may connect at a time, this can almost always be set to 1.

```
client_sock, address_and_port = sock.accept()
```

The method `accept` takes no parameters and returns two values. The first is a brand new `BluetoothSocket` object connected to the client. The second is a tuple, containing the address and port number of the connected client.

Outgoing Connections

Client programs use the `connect` method to establish an outgoing connection. They do not call `bind` or the other two server-specific functions:

```
connect( address_and_port )
```

The method `connect` requires a tuple specifying an address and port number, just as with the method `bind`, but the address must be a valid Bluetooth address. (It cannot be empty.)

Consider the server and client code in Examples 2.2 and 2.3. The server declares that all communication happens on RFCOMM port 1. After the server is waiting to accept a connection, the client tries to connect to the Bluetooth device with address "01:23:45:67:89:AB" on port 1.

Using a Connected Socket

Once a socket is connected, the `send` and `recv` methods can be used to send and receive data:

```
sock.recv( max_bytes_to_recv )
sock.send( data )
```

The function `recv` takes a parameter specifying the maximum amount of data to return, specified in bytes, and returns a sequence of bytes received on the connection. To send a packet of data over a connection, simply pass it to `send`, which queues it up for delivery. When an application is finished with its Bluetooth communications, it can disconnect by calling the `close` method on a connected socket:

```
sock.close()
```

When a socket is no longer connected, the `recv` method will always return an empty string. This is the only case where `recv` returns empty strings, which makes it a reliable way of determining when a connection has been terminated.

Error Handling

We've left out error-handling code in these examples for clarity, but the process is fairly straightforward. When any of the Bluetooth operations fail for some reason (e.g., connection timeout, no local Bluetooth resources are available, etc.) then a `BluetoothError` is raised with an error message indicating the reason for failure.

2.3 Communicating with L2CAP

(GNU / Linux Only)

The basics of using L2CAP as a transport protocol are demonstrated in Examples 2.4 and 2.5. Notice that using L2CAP sockets is almost identical to using RFCOMM sockets.

Example 2.4 l2cap-server.py

```
from bluetooth import *

port = 0x1001
backlog = 1

server_sock=BluetoothSocket( L2CAP )
server_sock.bind(("",port))
server_sock.listen(backlog)

client_sock,address = server_sock.accept()
print "Accepted connection from ",address

data = client_sock.recv(1024)
print "received [%s]" % data

client_sock.close()
server_sock.close()
```

Example 2.5 l2cap-client.py

```
from bluetooth import *

server_addr = "01:23:45:67:89:AB"
port = 0x1001

sock=BluetoothSocket(L2CAP)
sock.connect((server_addr, port))

sock.send("hello!!")

sock.close()
```

Aside from passing in `L2CAP` in place of `RFCOMM` as a parameter to the `BluetoothSocket` constructor, the only major difference is the choice of port number. Recall that L2CAP strictly limits port numbers to odd values between 4097 and 32,765. It is common to use hexadecimal notation when referring to L2CAP port numbers, just because they tend to look a little cleaner.

Maximum Transmission Unit

As a datagram-based protocol, packets sent on L2CAP connections have an upper size limit. Although the default value of 672 bytes is small, it can be adjusted. Each device at the end point of a connection maintains an *incoming maximum transmission unit* (MTU), which specifies the maximum size packet it can receive. If both devices adjust their incoming MTU settings, then it is possible to change the MTU of the entire connection beyond the 672-byte default up to 65,535 bytes and as low as 48 bytes. In PyBluez, the `set_l2cap_mtu` function is used to adjust this value. For example,

```
set_l2cap_mtu( l2cap_sock, new_mtu )
```

This method is fairly straightforward, and takes two parameters. The first, parameter 12cap_sock, should be a connected L2CAP `BluetoothSocket`. The second, `new_mtu`, should be an integer specifying the incoming MTU for the local adapter. Calling this function affects only the specified socket, and does not change the MTU for any other socket. Here's an example of how we might use it to raise the MTU:

```
l2cap_sock = BluetoothSocket( L2CAP )
    .
    . # connect the socket. This must be done before setting the MTU!
    .
set_l2cap_mtu( l2cap_sock, 65535 )
```

Remember that both devices involved in a connection should raise their MTU settings. It is not illegal for each side to have a different MTU, but that just gets confusing.

Best-Effort Transmission

Despite our expressed reservations about using best-effort L2CAP channels in 1.2.2, there are situations where best-effort semantics are preferred to reliable semantics. For example, when sending time-critical data such as audio or video data, it may be more important to forget about a few bad packets and keep sending at a constant data rate so that the connection doesn't "skip." Adjusting the reliability semantics of a connection in PyBluez is also a simple task, and can be done with the `set_packet_timeout` function:

```
set_packet_timeout( bluetooth_address, timeout )
```

The function `set_packet_timeout` takes a Bluetooth address and a timeout, specified in milliseconds, as input and tries to adjust the packet timeout for all L2CAP and RFCOMM connections to that device. The process must have superuser privileges, and there must be an active connection to the specified address. The effects of adjusting this parameter will last as long as any active connections are open, including those which outlive the Python program. If all connections to the specified Bluetooth device are closed and new ones are reestablished, then the connection reverts to the default of never timing out.

2.4 Service Discovery Protocol

So far we've seen how to detect nearby Bluetooth devices and establish the two main types of data transport connections, all using fixed Bluetooth addresses and port numbers that were determined at design time. A robust Bluetooth application service uses dynamically allocated port numbers. Consequently, client applications must have a way to determine the port number for the service. After all, what's the point of having a server running on a random port if the clients can't find it? Here, we'll see how to use the Service

Discovery Protocol (SDP) for this purpose. To get started, Examples 2.6 and 2.7 show the RFCOMM client and server from Section 2.2 modified to use dynamic port numbers and SDP.

Example 2.6 rfcomm-server-sdp.py

```
from bluetooth import *

server_sock=BluetoothSocket( RFCOMM )
server_sock.bind(("",PORT_ANY))
server_sock.listen(1)

advertise_service( server_sock, "SampleServer",
                   service_classes = [ SERIAL_PORT_CLASS ],
                   profiles = [ SERIAL_PORT_PROFILE ] )

client_sock, client_info = server_sock.accept()
print "Accepted connection from ", client_info

client_sock.send("PyBluez server says Hello!!")
data = client_sock.recv(1024)
print "received: ", data

client_sock.close()
server_sock.close()
```

Example 2.7 rfcomm-client-sdp.py

```
import sys
from bluetooth import *

service_matches = find_service( name = "SampleServer",
                                uuid = SERIAL_PORT_CLASS )

if len(service_matches) == 0:
    print "couldn't find the service!"
    sys.exit(0)

first_match = service_matches[0]
```

```
port = first_match["port"]
name = first_match["name"]
host = first_match["host"]

print "connecting to ", host

sock=BluetoothSocket( RFCOMM )
sock.connect((host, port))
sock.send("PyBluez client says Hello!!")
data = sock.recv(80)
print "received: ", data
sock.close()
```

Notice that these examples are very similar to the previous examples in Section 2.2. However, instead of hard-coding a port number, the server dynamically allocates a port number. A socket is first bound and set to listening mode, before the server advertises an SDP service. The code then continues as in the previous examples. The client searches for a service record and uses that information to establish a connection to the appropriate port. The concept is simple: the server advertises a service, and the client searches for and finds that service. Actual port numbers are allocated automatically.

Dynamically Allocating Port Numbers

Binding a socket to a dynamically assigned port is quite simple. Just use the constant `PORT_ANY` instead of a port number when invoking `bind`. For example,

```
sock = BluetoothSocket( RFCOMM )
sock.bind( ("", PORT_ANY) )
```

This method can be used for both RFCOMM and L2CAP sockets. If no ports are available, then `bind` raises an `IOError` exception.

To determine which port the socket was actually bound to, use the `getsockname` method:

```
localaddr, localport = sock.getsockname()
```

This method returns the address and port of the local Bluetooth adapter bound by the socket, and can only be used after a call to `bind`. Note that if

the address of a local Bluetooth adapter was not specified during the initial call to `bind`, then the address returned by `getsockname` may be "00:00:00:00:-00:00."

Advertising a Service

Once an application has a bound and listening socket, it can advertise a service with the local SDP server. This is done with the `advertise_service` function:

```
advertise_service( sock, name, service_id="", service_classes=[],
                   profiles=[], provider="", description="" )
```

Only the first two parameters to this function, `sock` and `name`, are required, and the rest have empty defaults.

`sock`: A `BluetoothSocket` object that must already be bound and listening.

`name`: A short text string describing the name of the service.

`service_id` Optional: The service ID of the service, specified as a string of the form "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX," where each "X" is a hexadecimal digit.

`service_classes` Optional: A list of service class IDs, each of which can be specified as a full 128-bit UUID in the form "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX," or as a reserved 16-bit UUID in the form "XXXX." A number of predefined UUIDs can be used here, such as `SERIAL_PORT_CLASS` or `BASIC_PRINTING_CLASS`. See the PyBluez documentation for a full list of predefined service class IDs.

`profiles` Optional: A list of profiles. Each item of the list should be a `( uuid, version )` tuple. A number of predefined profiles can be used here, such as `SERIAL_PORT_PROFILE` or `LAN_ACCESS_PROFILE`. See the PyBluez documentation for a full list of predefined profiles.

`provider` Optional: A short text string describing the provider of the service.

`description` Optional: A short text string describing the actual service.

The function `advertise_service` registers a service record with the local SDP server. To unregister the service, use the function `stop_advertising`:

```
stop_advertising( sock )
```

This function takes a single parameter, `sock`, which is the socket originally used to advertise the service. Another way to unregister a service is to simply close the socket, which will automatically trigger a call to `stop_advertising`.

Searching and Browsing Services

The function `find_service` can find a single service, or a list of services on one or more nearby Bluetooth devices:

```
results = find_service( name = None, uuid = None, address = None )
```

With no arguments, the function `find_service` returns a listing of all services offered by all nearby Bluetooth devices. It first scans for nearby devices and then queries each for a list of its services; if there are many devices in range, this could take a long time. Three optional parameters control and filter the search results:

`name`: Restricts the search to services with this name.
`uuid`: Restricts the search to services with any attribute value matching this `uuid`. Note that the matching UUID could be either the service ID, an entry in the service class ID list, an entry in the profiles list, or any other UUID that appears in the service record somewhere.
`address`: Only searches the Bluetooth device with this `address`. In the special case in which this is "`localhost`," then the local SDP server is searched.

The results of this search is a list of dictionary objects. Each dictionary has nine keys, which describe the corresponding service. The value for a key may be `None`, which indicates that it was not specified in the service record. The keys and their values correspond to the parameters of the advertised service and they are

`host`: The Bluetooth address of the device advertising the service.
`name`: The name of the service being advertised.
`description`: A description of the service.
`provider`: The provider of the service.
`protocol`: A text string indicating which transport protocol the service is using. This can take on one of three values: "`RFCOMM`," "`L2CAP`," or "`UNKNOWN`."
`port`: If `protocol` is either "`RFCOMM`" or "`L2CAP`," then this is an integer indicating which port number the service is running on.

`service-classes`: A list of service class IDs, in the same format as used for `advertise_service`.
`service-id`: The service ID of the advertised service.
`profiles`: A list of profiles, in the same format as used for `advertise_service`.

2.5 Advanced Usage

Although the techniques described in this chapter so far should be sufficient for most Bluetooth applications with simple and straightforward requirements, some applications may require more advanced functionality or finer control over the Bluetooth system resources. This section describes asynchronous Bluetooth communications and a few GNU/Linux-specific extensions.

Asynchronous Socket Programming with `select`

Examples 2.8 and 2.9 demonstrate how to use `select` for asynchronous socket programming.

Example 2.8 rfcomm-server-select.py

```
from bluetooth import *
from select import *

server_sock = BluetoothSocket( RFCOMM )
server_sock.setblocking(False)
server_sock.bind(("", 3))

while True:
    print "waiting for connection"
    readable, writable, excepts = select( [server_sock], [], [], 1 )
    if server_sock in readable:
        client_sock, client_info = server_sock.accept()
        client_sock.setblocking(False)
        print "Accepted connection from ", client_info
        break

while True:
    print "waiting for data"
    readable, writable, excepts = select( [client_sock], [], [], 1 )
    if client_sock in readable:
        data = client_sock.recv(1024)
```

```
        print "received [%s]" % data
        break

client_sock.close()
server_sock.close()
```

Example 2.9 rfcomm-client-select.py

```
from bluetooth import *
from select import *

sock=BluetoothSocket( RFCOMM )
sock.setblocking(False)

try: sock.connect(("01:23:45:67:89:AB", 3))
except: pass

while True:
    print "waiting for connection"
    readable, writable, excepts = select( [], [sock], [], 1 )
    if sock in writable:
        sock.send("hello!!")
        sock.close()
        break
```

The first step is to switch each socket to nonblocking mode, so that operations that normally block return immediately instead. To switch a socket into nonblocking mode, use the `setblocking` method:

```
sock = BluetoothSocket( RFCOMM )
sock.setblocking( False )
```

In the server example, the client socket is explicitly switched to nonblocking mode. This is important to do, as the blocking settings of a server socket are not transferred to the client sockets it spawns:

```
client_sock, client_info = server_sock.accept()
client_sock.setblocking(False)
```

In the client example, the `connect` method is guarded by a `try: ... except:` block, because `connect` will always fail and throw an exception when invoked on a nonblocking socket. This may seem odd at first, but is intended to be

Table 2.2 `select` events.

Event	List
Outgoing connection established (client)	Write
Data received on socket	Read
Incoming connection accepted (server)	Read
Can send data (i.e., send buffer not full)	Write
Disconnected	Read

consistent with the notion that `connect` always fails unless the connection has been successfully established:

```
try: sock.connect(("01:23:45:67:89:AB", 3))
except: pass
```

Once a socket is in nonblocking mode, it can safely be used with the `select` module without fear of it blocking at an inappropriate time. The `select` module comes as part of the standard Python distribution, and provides the `select` function:

```
from select import *
results = select( to_read, to_write, to_exc, [timeout] )
readable, writable, excepts = results
```

`select` can wait for three different types of events – read events, write events, and exceptions. The first three parameters are lists of objects; which list an object is in determines which type of event `select` will detect for that object. An object can be in multiple lists. As soon as `select` detects an event, it returns three lists, each of which contains objects from the original lists where event activity was detected. The fourth parameter to `select` is optional and specifies a timeout as a floating point number in seconds. If no events are detected before the timeout elapses, then `select` returns three empty lists. Table 2.2 summarizes which list to put a socket in for detecting specific events.

You'll notice a couple of things here. First, the third list for exceptions isn't used at all. `select` is meant to be used for all different types of objects, and the third list is used elsewhere, just not in Bluetooth. Second, we didn't mention searching for nearby devices or SDP. We'll talk about the device discovery process next, but unfortunately there aren't yet any asynchronous techniques for SDP. In this case, you'll have to rely on threads for nonblocking operations, but hopefully that will change in the future.

2.5.1 Asynchronous Device Discovery

(GNU / Linux Only)

Asynchronously searching for nearby devices and determining their user-friendly names can also be done with `select`, but is a bit more complicated and involves the use of a new class, the `DeviceDiscoverer`. Example 2.10 shows an example of how to use `select` and `DeviceDiscoverer` for this purpose.

To asynchronously detect nearby Bluetooth devices, create a subclass of `DeviceDiscoverer` and override the `pre_inquiry`, `device_discovered`, and `inquiry_complete` methods. To start the discovery process, invoke `find_devices`, which returns immediately. `pre_inquiry` is called immediately before the actual inquiry process begins.

Example 2.10 asynchronous-inquiry.py

```
from bluetooth import *
from select import *

class MyDiscoverer(DeviceDiscoverer):
    def pre_inquiry(self):
        self.done = False
    def device_discovered(self, address, device_class, name):
        print "%s - %s" % (address, name)

    def inquiry_complete(self):
        self.done = True

d = MyDiscoverer()
d.find_devices(lookup_names = True)

while True:
    can_read, can_write, has_exc = select( [d], [], [] )

    if d in can_read:
        d.process_event()

    if d.done: break
```

Call `process_event` to have the `DeviceDiscoverer` process pending events, which can be either a discovered device or the inquiry completion. When a nearby device is detected, `device_discovered` is invoked, with the address and device class of the detected device. If `lookup_names` was set in the call to `find_devices`, then `name` will also be set to the user-friendly name of the device. For more information about device classes, see the Bluetooth web site.* The `DeviceDiscoverer` class can be used directly with the `select` module.

2.5.2 The `_bluetooth` Module

(GNU / Linux Only)

The `bluetooth` module provides classes and utility functions useful for the most common Bluetooth programming tasks. More advanced functionality can be found in the `_bluetooth` extension module, which is little more than a thin wrapper around the BlueZ C API described in the next chapter. Lower level Bluetooth operations, such as establishing a connection with the actual Bluetooth microcontroller on the local machine and reading signal strength information, can be performed with the `_bluetooth` module in most cases without having to resort to the C API.

HCI Sockets

An HCI socket, created by calling the `hci_open_dev` function, represents a direct connection to the microcontroller on a local Bluetooth adapter. This allows complete control over almost all Bluetooth functionality that the adapter has to offer, and is often useful for low-level tweaking:

```
hci_sock = hci_open_dev( [ adapter_number ] )
```

The function takes a single optional parameter, specifying which local Bluetooth adapter to use. The first Bluetooth adapter is 0, the second is 1, and so on. If you don't care which one to use (or if you only have a single Bluetooth adapter), then you can leave this out.

Communicating with the microcontroller consists of sending commands and receiving events. A command is composed of three parts – an *Opcode Group Field* (OGF), an *Opcode Command Field* (OCF), and the command parameters, which are different for each command. The OGF specifies the

* https://www.bluetooth.org/foundry/assignnumb/document/baseband

general category of command, such as device control or link control. The OCF specifies the exact command within the OGF category. There are dozens of combinations that can be used here, all of which are neatly laid out in the Bluetooth specification.

Most operations will have a *request-reply* format, where an event is generated by the microcontroller immediately after the command. This event contains the result of the command (the microcontroller's reply to the user's request), and typically indicates whether the command succeeded or not along with relevant information. Operations that follow this format can be performed using the `hci_send_req` function:

```
reply = hci_send_req( hci_sock, ogf, ocf, event, reply_len,
                      [params], [timeout] )
```

The first three parameters to this function are the HCI socket to use, and the OGF and OCF of the command. `event` specifies the type of event to wait for, and `reply_len` specifies the size of the reply packet, in bytes, to expect from the microcontroller. `params` is optional because some commands don't take any parameters, and if specified should be a packed binary string. `timeout`, also optional, specifies in milliseconds how long to wait for the request to complete. The function returns an unprocessed binary string containing the microcontroller's reply.

As with the OGF and OCF fields, the exact details on how to pack the parameters, which event to wait for, and how to interpret the reply are all defined in the Bluetooth specification.

2.6 SCO Audio Sockets

(GNU / Linux Only)

Establishing and using a synchronous connection oriented (SCO) audio socket to transfer voice-quality audio can be done in PyBluez by creating a `BluetoothSocket` of type `SCO`. Examples 2.11 and 2.12 demonstrate how to do this.

Voice Setting

Audio data transmitted over an SCO connection is always encoded with either a log pulse-code modulation (PCM) or continuously variable slope delta (CVSD) codec prior to the actual transmission, and decoded accordingly when received. This encoding/decoding step happens on the Bluetooth

microcontroller, and a software application needs to only instruct the microcontroller which codec to use. CVSD is by far the most widely used codec,

Example 2.11 sco-server.py

```
import struct
from bluetooth import *
import _bluetooth as bt

# check voice settings.  switch to S16 LE mono 8kHz CVSD if needed
hci_sock=bt.hci_open_dev()
response = bt.hci_send_req(hci_sock, bt.OGF_HOST_CTL,
        bt.OCF_READ_VOICE_SETTING, bt.EVT_CMD_COMPLETE, 3, "")
status, voice_setting = struct.unpack("<BH", response)
if voice_setting != 0x60:
    new_vs = struct.pack("<H", 0x60)
    bt.hci_send_req(hci_sock, bt.OGF_HOST_CTL,
            bt.OCF_WRITE_VOICE_SETTING, bt.EVT_CMD_COMPLETE, 1, new_vs)

server_sock=BluetoothSocket( SCO )
server_sock.bind(("",))
server_sock.listen(1)

client_sock, client_info = server_sock.accept()
print "Accepted connection from ", client_info

# receive 1 second's worth of audio (data is 16-bit mono 8 kHz audio)
data = ""
while len(data) < 16000:
    data = data + client_sock.recv(16000 - len(data))
    print len(data)

client_sock.close()
server_sock.close()
```

Example 2.12 sco-client.py

```
import struct
import time
from bluetooth import *
import _bluetooth as bt

# check voice settings.  switch to S16 LE mono 8kHz CVSD if needed
hci_sock=bt.hci_open_dev()
```

```
response = bt.hci_send_req(hci_sock, bt.OGF_HOST_CTL,
        bt.OCF_READ_VOICE_SETTING, bt.EVT_CMD_COMPLETE, 3)
status, voice_setting = struct.unpack("<BH", response)
if voice_setting != 0x60:
    new_vs = struct.pack("<H", 0x60)
    bt.hci_send_req(hci_sock, bt.OGF_HOST_CTL,
            bt.OCF_WRITE_VOICE_SETTING, bt.EVT_CMD_COMPLETE, 1, new_vs)

# determine the maximum packet size
response = bt.hci_send_req(hci_sock, bt.OGF_INFO_PARAM,
        bt.OCF_READ_BUFFER_SIZE, bt.EVT_CMD_COMPLETE, 8)
status, acl_mtu, sco_mtu, acl_nbufs, sco_nbufs = \
        struct.unpack("<BHBHH", response)

server_address = "01:23:45:67:89:AB"

sock=BluetoothSocket( SCO )
sock.connect((server_address,))

# send one second's worth of silence
tosend = "\0" * 16000
sent = 0
while sent < 16000:
    sent += sock.send(tosend[sent:sent+sco_mtu])
    print sent

time.sleep(2)

sock.close()
```

since it is the default for the Bluetooth Headset Profile. In the examples, the first step taken by both the client and the server is to configure the local adapter to use the CVSD codec. PyBluez does not provide a convenience function to do this, so an HCI socket is used to directly communicate with the microcontroller. First, the application checks the existing settings to determine if they should be changed:

```
hci_sock=bt.hci_open_dev()
response = bt.hci_send_req(hci_sock, bt.OGF_HOST_CTL,
        bt.OCF_READ_VOICE_SETTING, bt.EVT_CMD_COMPLETE, 3, "")
status, voice_setting = struct.unpack("<BH", response)
```

Volume 2, Section 6.12, of the Bluetooth 2.0 specification describes how to interpret the audio settings for SCO connections. A value of `0x60` corresponds to mono signed 16-bit little-endian audio using the CVSD, and is the format used by Bluetooth headsets. Sections 7.3.29 and 7.3.30 of the same volume describe how to format an HCI packet to check and adjust this setting. Next, if necessary, the application makes the change:

```
if voice_setting != 0x60:
   new_vs = struct.pack("<H", 0x60)
   bt.hci_send_req(hci_sock, bt.OGF_HOST_CTL, bt.OCF_WRITE_VOICE_SETTING,
           bt.EVT_CMD_COMPLETE, 1, new_vs)
```

SCO Maximum Transmission Unit

As with other datagram-based protocols, SCO packets have a fixed maximum transmission size. Audio data must be divided into chunks smaller than or equal to this size before being sent over an SCO connection. Unlike the L2CAP MTU, the SCO MTU cannot be adjusted. To determine this size, we can consult Section 7.4.5 of the Bluetooth 2.0 specification and then use the Read_Buffer_Size HCI command:

```
response = bt.hci_send_req(hci_sock, bt.OGF_INFO_PARAM,
        bt.OCF_READ_BUFFER_SIZE, bt.EVT_CMD_COMPLETE, 8)
status, acl_mtu, sco_mtu, acl_nbufs, sco_nbufs = \
        struct.unpack("<BHBHH", response)
```

This command also provides information on the maximum packet size for ACL connections and the maximum number of packets the Bluetooth microcontroller can buffer. Neither of these are required to use SCO sockets.

Using an SCO Socket

Once the audio settings have been configured, actually setting up and using an SCO socket is almost trivially simple. First, instantiate a `BluetoothSocket` and pass in `SCO`.

```
sock=BluetoothSocket(SCO)
```

Listening SCO sockets must be bound to a local adapter. Binding to a port number is not necessary, since there are no port numbers in SCO. As before, specifying an empty string as the address indicates that any local adapter is fine:

```
server_sock.bind(("",))
```

Note the odd syntax here. Even though there is only one parameter (the local address), it still must be enclosed within a tuple.

To establish an outgoing connection, invoke `connect` and pass it a tuple containing the address of the target device.

```
sock.connect((server_address,))
```

The rest – accepting connections, transmitting data, and terminating the connection – is the same as before.

Although we've mentioned it before, it is important to stress that SCO sockets should never be used to transmit data transparently. Although the audio codecs increase the robustness of audio data to noise and packet loss, they also mangle the data so that any given bits transmitted from one side of a connection do not come out as the same bits when received; they sound the same, but do not look the same.

2.7 Obtaining and Installing PyBluez

PyBluez is distributed at `http://org.csail.mit.edu/pybluez` under the terms of the GNU General Public License. The current version as of this writing is PyBluez 0.8, released on July 31, 2006.

Pre-Compiled Binary Installers

Windows XP: A Windows XP installer can be downloaded from the PyBluez Web site.

Debian Linux: PyBluez is available starting in Etch as the package `python-bluez`. Use `apt-get` to install

```
# apt-get update
# apt-get install python-bluez
```

Ubuntu Linux: PyBluez is available starting with Ubuntu Edgy Eft as the package `python-bluez`. As with Debian, use `apt-get to install`.

Fedora Linux: PyBluez is available starting in Core 6 Extras as the package `pybluez`. Use `yum to install`

```
# yum install pybluez
```

Other distributions of Linux require downloading and compiling the Python extension module from source.

Building from Source

The most recent source code can be directly downloaded from the PyBluez Web site. Instructions for compiling and installing PyBluez are given here, although the most accurate and up-to-date information is always on the Web site.

GNU/Linux build requirements are as follows:

- Python 2.3 or more recent version,
- BlueZ libraries and header files (`http://www.bluez.org`), and
- GCC.

Windows XP build requirements are as follows:

- Python 2.3 or more recent version,
- Microsoft Visual Studio,
- Microsoft Windows Platform SDK, and
- Microsoft Windows XP SP1 or more recent version.

Build instructions: Unpack the PyBluez distribution and navigate to the unpacked directory in a command shell. Then invoke Python to compile and install the extension module:

```
# python setup.py install
```

2.8 Summary

Bluetooth programming in Python is simple and straightforward with the PyBluez extension module. In this chapter, we've focused on seeing how the concepts and techniques introduced in Chapter 1 transfer to an actual programming language. Python excels as a language with which to learn Bluetooth programming, as well as for prototyping and developing applications with simple Bluetooth requirements. Applications that require access to more advanced Bluetooth features are better served with a different development environment, such as those introduced in the coming chapters.

Device discovery	`int dev_id = hci_get_route(NULL);` `hci_inquiry(dev_id, ...);`
Name lookup	`int hci_sock = hci_open_dev(dev_id);` `hci_read_remote_name(hci_sock, ...);`
SDP connect	`sdp_session_t *session;` `session = sdp_connect(BDADDR_ANY, ...);`
SDP search	`sdp_service_search_attr_req(session, ...);`
Establish an outgoing connection	`int s = socket(AF_BLUETOOTH, type, proto);` `connect(s, ...);`

Establish an incoming connection	`int s = socket(AF_BLUETOOTH, type, proto);`
	`bind(s, ...);`
	`listen(s, backlog);`
	`accept(s, ...);`

Advertise an SDP service	`sdp_session_t *sess;`
	`sess = sdp_connect(BDADDR_ANY,`
	`BDADDR_LOCAL, ...);`
	`sdp_record_register(sess, ...);`

Transfer data	`send(s, data, datalen, flags);` `recv(s, buffer, buflen, flags);`
Disconnect	`close(s);`

BlueZ Quick Reference

Chapter 3

C Programming With GNU/Linux

This chapter explains how to write Bluetooth communication code in the C programming language. There are reasons to prefer developing in C instead of in a high-level language, such as Python. The Python environment might not be available or might not fit on the target device; strict application requirements on program size, speed, and memory usage may preclude the use of an interpreted language like Python; the programmer may desire finer control over the local Bluetooth adapter than PyBluez provides; or the project may be to create a shared library for other applications to link against instead of a stand-alone application. As of this writing, BlueZ is a powerful Bluetooth communications stack with extensive APIs that allow a user to fully exploit all local Bluetooth resources. It is open source, freely available, and comes standard with all major distributions of GNU/Linux.

This chapter presents a short introduction to developing Bluetooth applications in C with BlueZ. The tasks introduced in the introduction and covered in Chapter 2 are now explained in greater detail here for C programmers. The steps and logic are the same; this chapter tries to unravel the tangled data structures needed to communicate with the Bluetooth protocol stack.

Table 3.1 illustrates the transport protocols supported by the BlueZ stack. Notice that all the entries are shaded, as BlueZ exposes all the essential APIs addressed in this text. The (RFCOMM), (L2CAP), and (SCO) protocols are accessed using the standard socket interface, while (HCI) is more easily used

Table 3.1 Transport protocols supported by BlueZ.

BlueZ	RFCOMM	L2CAP	SCO	HCI

with a number of convenience functions that provide wrappers around the various HCI commands and events (see also, Chapter 7). Although not documented in this chapter, BlueZ support for the OBEX protocol for transferring files is available via the OpenOBEX libraries.

3.1 Choosing a Communication Target

We begin by showing how to choose a communication target. A simple program that detects nearby Bluetooth devices is shown in Example 3.1. The program reserves system Bluetooth resources, scans for nearby Bluetooth devices, and then looks up the user-friendly name for each detected device. A more detailed explanation of the data structures and functions used follows.

Example 3.1 simplescan.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>

int main(int argc, char **argv)
{
    inquiry_info *devices = NULL;
    int max_rsp, num_rsp;
    int adapter_id, sock, len, flags;
    int i;
    char addr[19] = { 0 };
    char name[248] = { 0 };
```

```
    adapter_id = hci_get_route(NULL);
    sock = hci_open_dev( adapter_id );
    if (adapter_id < 0 || sock < 0) {
        perror("opening socket");
        exit(1);
    }

    len  = 8;
    max_rsp = 255;
    flags = IREQ_CACHE_FLUSH;
    devices = (inquiry_info*)malloc(max_rsp * sizeof(inquiry_info));

    num_rsp = hci_inquiry(adapter_id, len, max_rsp, NULL, &devices,
            flags);
    if( num_rsp < 0 ) perror("hci_inquiry");

    for (i = 0; i < num_rsp; i++) {
        ba2str(&(devices+i)->bdaddr, addr);
        memset(name, 0, sizeof(name));
        if (0 != hci_read_remote_name(sock, &(devices+i)->bdaddr,
                    sizeof(name), name, 0)) {
            strcpy(name, "[unknown]");
        }
        printf("%s  %s\n", addr, name);
    }

    free( devices );
    close( sock );
    return 0;
}
```

Compiling the Example

To compile our program, invoke `gcc` and link against `libbluetooth`:

```
# gcc -o simplescan simplescan.c -lbluetooth
```

Representing Bluetooth Addresses

```
typedef struct {
    uint8_t b[6];
} __attribute__((packed)) bdaddr_t;
```

The basic data structure used to specify a Bluetooth device address is a packed array of 6 bytes, and referred to as `bdaddr_t`. All Bluetooth addresses in BlueZ will be stored and manipulated as `bdaddr_t` structures. Two convenience functions, `str2ba` and `ba2str`, can be used to convert between strings and `bdaddr_t` structures.

```
int str2ba( const char *str, bdaddr_t *ba );
int ba2str( const bdaddr_t *ba, char *str );
```

The function `str2ba` takes a string of the form "XX:XX:XX:XX:XX:XX," where each XX is a hexadecimal number specifying 1 byte of the 6-byte address, and packs it into a `bdaddr_t`. The function `ba2str` does exactly the opposite.

Choosing and Opening a Local Bluetooth Adapter

Local Bluetooth adapters are assigned identifying numbers starting with 0, and a program must specify which adapter to use when allocating system resources. Usually, there is only one adapter or it doesn't matter which one is used, so passing `NULL` to `hci_get_route` will retrieve the resource number of the first available Bluetooth adapter:

```
int hci_get_route( bdaddr_t *addr );
```

This function actually returns the resource number of any adapter whose Bluetooth address does not match the one passed in as a parameter, so by passing in `NULL`, the program essentially asks for any available adapter.

If there are multiple Bluetooth adapters present, and we know which one we want, then we can use `hci_devid`:

```
int hci_devid( const char *addr );
```

Unlike its counterpart, `hci_devid` returns the resource number of the Bluetooth adapter whose address matches the one passed in as a parameter. This is one of the few places where a BlueZ function uses a string representation to work with a Bluetooth address instead of a `bdaddr_t` structure.

> *Note:* Some of the functions here use "dev" (short for "device") in their names, while we've been using the term "adapter." In general, we use *device* to refer to any machine capable of Bluetooth communication, while *adapter* refers specifically to the local (machine executing the program) Bluetooth device. BlueZ does not always make this distinction.

Once the program has chosen which adapter to use in scanning for nearby devices, it must allocate resources to use that adapter. This can be done with the `hci_open_dev` function:

```
int hci_open_dev( int adapter_id );
```

To be more specific, this function opens a socket connection to the microcontroller on the specified local Bluetooth adapter. Keep in mind that this is *not* a connection to a remote Bluetooth device, and is used specifically for controlling the local adapter. Later on, in Section 3.5, we'll see how to use this type of socket for more advanced Bluetooth operations, but for now we'll just be using it for the device inquiry process. The result returned by `hci_open_dev` is a handle to the socket. On error, it returns −1 and sets `errno`.

> *Note:* Although tempting, it is *not* a good idea to hard-code the device number 0, because that is not always the ID of the first adapter. For example, if there were two adapters on the system and the first adapter (ID 0) is disabled, then the first *available* adapter is the one with ID 1.

Device Discovery

When an inquiry message is broadcast, nearby Bluetooth devices respond by sending their Bluetooth address. Of course, devices must be in the correct mode in order to respond, and even then they may not hear the inquiry or take too long to respond. We often say that our program "scans" for nearby devices, but it is actually scanning for devices that responded to the inquiry. A Bluetooth device discovery operation is initiated by the function `hci_inquiry`, which issues the inquiry signal and returns a list of devices that respond along

with some of their basic information placed in the variable `devices` that is passed as a parameter to the function:

```
int hci_inquiry(int adapter_id, int len, int max_rsp,
            const uint8_t *lap, inquiry_info **devs, long flags);
```

A quick look at the parameters shows that the function does not actually use the socket opened in the previous step. Rather, `hci_inquiry` takes the resource number returned by `hci_get_route` (or `hci_devid`) as its first parameter, and it creates its own internal socket. Nearly all other functions covered in this chapter will use the socket opened by `hci_open_dev`.

The parameters control both the scanning and return information. The inquiry lasts for at most 1.28 * `len` seconds, and at most `max_rsp` responding devices will be returned in the output parameter `devs`, which must be large enough to accommodate `max_rsp` results. We suggest using a `max_rsp` of 255 for a standard 10.24-s inquiry. The `devs` parameter is an array of `inquiry_info` structures:

```
typedef struct {
    bdaddr_t    bdaddr;
    uint8_t     pscan_rep_mode;
    uint8_t     pscan_period_mode;
    uint8_t     pscan_mode;
    uint8_t     dev_class[3];
    uint16_t    clock_offset;
} __attribute__ ((packed)) inquiry_info;
```

The first entry, the `bdaddr` field, is the most useful and gives the address of the detected device. Of the remaining fields, the `dev_class` field may also be of interest, as it conveys general information about the type of device detected (if it's a printer, phone, computer, etc.) and the services offered (file transfer, audio, network access, etc.). The exact formatting of this field is described in the "Assigned Numbers – Bluetooth Baseband" document, distributed on the `bluetooth.org` Web site.* The remaining fields are used for low-level communication, and are not usually useful. If you're really interested, all the gory details can be found in the Bluetooth specification.

The final parameter, `flags`, indicates whether or not to use previously discovered device information or to start afresh. If it is set to `IREQ_CACHE_FLUSH`,

* https://www.bluetooth.org/foundry/assignnumb/document/baseband

then the cache of previously detected devices is flushed before performing the current inquiry. If `flags` is set to 0, then the results of previous inquiries may be returned, even if the devices are no longer in range.

Name Lookup

Given the list of the addresses of nearby Bluetooth device, it is common practice to then determine their user-friendly names. The `hci_read_remote_name` function is used for this purpose:

```
int hci_read_remote_name(int hci_sock, const bdaddr_t *addr,
                         int len, char *name, int timeout);
```

This function attempts, for at most `timeout` milliseconds, to use the socket `hci_sock` in order to query the device with Bluetooth address `addr` for its user-friendly name. On success, it returns 0 and the first `len` bytes of the device's user-friendly name stored in the supplied character array: `name`.

The function `hci_read_remote_name` only tries to resolve a single name, so a program will typically invoke it many times to get a list of all the user-friendly names of nearby Bluetooth devices.

Error Handling

So far, all the functions introduced return an integer on completion. If the function succeeds in doing whatever the program requested, then the return value is always 0. If the function fails, then the return value is −1 and the `errno` global variable is set to indicate the type of error. This is true of all the `hci_` functions, as well as for almost all of the socket functions described in the next few sections. (The exceptions are `socket`, `send`, and `recv`.)

In the examples, we've left out error checking for clarity, but a robust program should examine the return value of each function call to check for potential failures. A simple way to incorporate error handling is to use the `strerror` function to print out what went wrong, and then exit. For example, consider the following snippet of code:

```
int dev_id = hci_get_route( NULL );
if( dev_id < 0 ) {
  fprintf(stderr, "error code %d: %s\n", errno, strerror(errno));
  exit(1);
}
```

If we ran this bit of code on a machine that does not have a Bluetooth adapter, we might see the following output:

```
error code 19: No such device
```

This might not be the best error message to show an actual user, but it should give you an idea of how to add error handling to your Bluetooth programs. For more information about using `errno`, consult a book on Linux programming.

3.2 RFCOMM Sockets

As with Python, establishing and using RFCOMM connections boil down to the standard Internet socket programming techniques, which we reviewed earlier in Section 1.2.5. To get started, the code in Examples 3.2 and 3.3 shows how to establish a connection using an RFCOMM socket, transfer some data, and disconnect. For simplicity, the client is hard coded to connect to a device with a Bluetooth address of "`01:23:45:67:89:AB`."

A careful reader of the previous chapter will notice that the examples have the same flow and structure used by the corresponding Python examples in Section 2.2. The seasoned Internet programmer will notice that these two examples are almost exactly the same as TCP programming examples, although with some socket creation and the addressing structure differences.

Example 3.2 rfcomm-server.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>

int main(int argc, char **argv)
{
    struct sockaddr_rc loc_addr = { 0 }, rem_addr = { 0 };
    char buf[1024] = { 0 };
    int s, client, bytes_read;
    unsigned int opt = sizeof(rem_addr);

    // allocate socket
```

```
    s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

   // bind socket to port 1 of the first available bluetooth adapter
    loc_addr.rc_family = AF_BLUETOOTH;
    loc_addr.rc_bdaddr = *BDADDR_ANY;
    loc_addr.rc_channel = 1;
    bind(s, (struct sockaddr *)&loc_addr, sizeof(loc_addr));

    // put socket into listening mode
    listen(s, 1);

    // accept one connection
    client = accept(s, (struct sockaddr *)&rem_addr, &opt);

    ba2str( &rem_addr.rc_bdaddr, buf );
    fprintf(stderr, "accepted connection from %s\n", buf);
    memset(buf, 0, sizeof(buf));

    // read data from the client
    bytes_read = recv(client, buf, sizeof(buf), 0);
    if( bytes_read > 0 ) {
        printf("received [%s]\n", buf);
    }

    // close connection
    close(client);
    close(s);
    return 0;
}
```

Example 3.3 rfcomm-client.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>
```

```
int main(int argc, char **argv)
{
    struct sockaddr_rc addr = { 0 };
    int s, status;
    char dest[18] = "01:23:45:67:89:AB";

    // allocate a socket
    s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

    // set the connection parameters (who to connect to)
    addr.rc_family = AF_BLUETOOTH;
    addr.rc_channel = 1;
    str2ba( dest, &addr.rc_bdaddr );

    // connect to server
    status = connect(s, (struct sockaddr *)&addr, sizeof(addr));
    // send a message
    if( 0 == status ) {
        status = send(s, "hello!", 6, 0);
    }

    if( status < 0 ) perror("uh oh");

    close(s);
    return 0;
}
```

The first step is to allocate or create a socket:

```
int socket( int domain, int type, int protocol );
```

For RFCOMM sockets, the three parameters to the `socket` function call should always be `AF_BLUETOOTH`, `SOCK_STREAM`, and `BTPROTO_RFCOMM`. The first parameter, `AF_BLUETOOTH`, specifies that it should be a Bluetooth socket. The second, `SOCK_STREAM`, requests a socket with streams-based delivery semantics. The third, `BTPROTO_RFCOMM`, specifically requests an RFCOMM socket. The `socket` function creates the RFCOMM socket and returns an integer that is used as a handle to that socket.

Addressing Structures

A `struct sockaddr_rc` addressing structure is required in order to establish either an incoming or outgoing connection with another Bluetooth device. Like the `struct sockaddr_in` used in TCP/IP programming, the addressing structure specifies details for client sockets (which device and port to connect to), as well as for listening sockets (which adapter to use and which port to listen on):

```
struct sockaddr_rc {
    sa_family_t rc_family;
    bdaddr_t    rc_bdaddr;
    uint8_t     rc_channel;
};
```

The `rc_family` field specifies the addressing family of the socket, and will always be `AF_BLUETOOTH`. For an outgoing connection, the remaining two fields, `rc_bdaddr` and `rc_channel`, specify the Bluetooth address and port number, respectively. For a listening socket, `rc_bdaddr` specifies the address of the local Bluetooth adapter to use and `rc_channel` specifies the port number to listen on. If you don't care which local Bluetooth adapter to use for the listening socket, then you can use `BDADDR_ANY` to indicate that any local Bluetooth adapter is acceptable.

Establishing a Connection

Once created, a socket must be connected in order to be of any use. The procedure for doing this depends on whether the application is accepting incoming connections (server sockets), or whether it's creating outgoing connections (client sockets).

Client sockets are simpler, and the process only requires making a single call to the `connect` function:

```
int connect( int sock, const struct sockaddr *server_info,
             socklen_t infolen );
```

The first parameter, `sockfd`, should be a socket handle created by the `socket` function. The second parameter should point to a `struct sockaddr_rc` addressing structure filled in with the details of the server's address and port number. Remember that you'll have to cast it into a `struct sockaddr*` to avoid compiler errors. Finally, the last parameter should always be

`sizeof(struct sockaddr_rc)` for RFCOMM sockets. The `connect` function uses this information to establish a connection to the specified server and returns once the connection has been established, or an error occurred.

Server sockets are a bit more complicated and involve three steps instead of just one. After the server socket is created, it must be bound to a local Bluetooth adapter and port number with the `bind` function:

```
int bind( int sock, const struct sockaddr *info, socklen_t infolen );
```

The first parameter, `sock`, is the server socket created by `connect`. `info` should point to a `struct sockaddr_rc` addressing structure filled in with the local Bluetooth adapter to use, and which port number to use. Finally, `addrlen` should always be `sizeof( struct sockaddr_rc )`.

After the socket is successfully bound, it should be placed into listening mode by using the `listen` function:

```
int listen( int sock, int backlog );
```

In between the time an incoming Bluetooth connection is accepted by the operating system and the time that the server application actually takes control, there may be other incoming connection requests. These new ones are placed into a backlog queue. The `backlog` parameter specifies how big this queue should be. Usually, a value of 1 is fine.

Once these steps have completed, the server application is ready to accept incoming connections using the `accept` function:

```
int accept( int server_sock, struct sockaddr *client_info,
            socklen_t *infolen );
```

The `accept` function waits for an incoming connection and returns a brand new socket. The returned socket represents the newly established connection with a client, and is what the server application should use to communicate with the client. If `client_info` points to a valid `struct sockaddr_rc` structure, then it is filled in with the client's information. Additionally, `infolen` will be set to `sizeof( struct sockaddr_rc )`. The server application can then make another call to `accept` and accept more connections, or it can close the server socket when finished.

Using a Connected Socket

Assuming a socket is connected, using it to send and receive data is straightforward. The `send` function transmits data, the `recv` function waits for and receives incoming data, and the `close` function disconnects a socket:

```
ssize_t send( int sock, const void *buf, size_t len, int flags );
ssize_t recv( int sock, void *buf, size_t len, int flags );
int close( int sock );
```

Both functions take four parameters, the first being a connected Bluetooth socket. For `send`, the next two parameters should be a pointer to a buffer containing the data to send and the amount of the buffer to send, in bytes. For `recv`, the second two parameters should be a pointer to a buffer into which incoming data will be copied and an upper limit on the amount of data to receive. The last parameter, `flags`, should be set to 0 for normal operation in both `send` and `recv`.

`send` returns the number of bytes actually transmitted, which may be less than the amount requested. In that case, the program should just try again, starting from where `send` left off. As usual, when `send` returns, the buffer can be reused even though the data bytes may have yet to be sent and are sitting in some operating system buffer.

The function `recv` returns the number of bytes actually received, which may be less than the maximum amount requested. The special case where `recv` returns 0 indicates that the connection is broken and no more data can be transmitted or received.

Once a program is finished with a connected socket, calling `close` on the socket disconnects and frees the system resources used by that connection.

3.3 L2CAP Sockets

Using L2CAP sockets is quite similar to using RFCOMM sockets, with the major differences being in the addressing structure, the availability of a few more options to control, and more ports. The code in Examples 3.4 and 3.5 demonstrate how to establish an L2CAP channel and transmit a short string of data. For simplicity, the client is hard-coded to connect to "01:23:45:67:89:AB."

For simple usage scenarios, the primary differences between using RFCOMM sockets and L2CAP sockets are the parameters to the `connect`

function and the addressing structure. For `connect`, the first parameter should still be `AF_BLUETOOTH`, but the next two parameters should be `SOCK_SEQPACKET` and `BTPROTO_L2CAP`, respectively. `SOCK_SEQPACKET` is used to indicate a socket with reliable datagram-oriented semantics where packets are delivered in the order sent. `BTPROTO_L2CAP` simply specifies the L2CAP protocol.

Example 3.4 l2cap-server.c

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>

int main(int argc, char **argv)
{
    struct sockaddr_l2 loc_addr = { 0 }, rem_addr = { 0 };
    char buf[1024] = { 0 };
    int s, client, bytes_read;
    unsigned int opt = sizeof(rem_addr);

    // allocate socket
    s = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);

    // bind socket to port 0x1001 of the first available
    // bluetooth adapter
    loc_addr.l2_family = AF_BLUETOOTH;
    loc_addr.l2_bdaddr = *BDADDR_ANY;
    loc_addr.l2_psm = htobs(0x1001);

    bind(s, (struct sockaddr *)&loc_addr, sizeof(loc_addr));

    // put socket into listening mode
    listen(s, 1);

    // accept one connection
    client = accept(s, (struct sockaddr *)&rem_addr, &opt);
```

```
    ba2str( &rem_addr.l2_bdaddr, buf );
    fprintf(stderr, "accepted connection from %s\n", buf);
    // read data from the client
    memset(buf, 0, sizeof(buf));
    bytes_read = recv(client, buf, sizeof(buf), 0);
    if( bytes_read > 0 ) {
        printf("received [%s]\n", buf);
    }

    // close connection
    close(client);
    close(s);
        return 0;
}
```

Example 3.5 l2cap-client.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>

int main(int argc, char **argv)
{
    struct sockaddr_l2 addr = { 0 };
    int s, status;
    char dest[18] = "01:23:45:67:89:AB";

    if(argc < 2) {
        fprintf(stderr, "usage: %s <bt_addr>\n", argv[0]);
        return 1;
    }
    strncpy(dest, argv[1], 18);

    // allocate a socket
    s = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
    // set the connection parameters (who to connect to)
```

```
    addr.l2_family = AF_BLUETOOTH;
    addr.l2_psm = htobs(0x1001);
    str2ba( dest, &addr.l2_bdaddr );

    // connect to server
    status = connect(s, (struct sockaddr *)&addr, sizeof(addr));

    // send a message
    if( 0 == status ) {
        status = send(s, "hello!", 6, 0);
    }

    if( status < 0 ) perror("uh oh");

    close(s);
        return 0;
}
```

In all three functions, `connect`, `bind`, and `accept`, L2CAP sockets use the `struct sockaddr_l2` addressing structure. It differs only slightly from the `struct sockaddr_rc` used in RFCOMM sockets:

```
struct sockaddr_l2 {
    sa_family_t     l2_family;
    unsigned short  l2_psm;
    bdaddr_t        l2_bdaddr;
};
```

The first field, `l2_family`, should always be `AF_BLUETOOTH`. The `l2_psm` field specifies an L2CAP port number, and `l2_bdaddr` denotes the address of either a server to connect to, a local adapter and port number to listen on, or the information of a newly connected client, depending on context.

Byte Ordering

Since Bluetooth deals with the transfer of data from one machine to another, the use of a consistent byte ordering for multibyte data types is crucial. Unlike network byte ordering, which uses a big-endian format, Bluetooth

byte ordering is little-endian,* where the least significant bytes are transmitted first. BlueZ provides four convenience functions to convert between host and Bluetooth byte orderings:

```
unsigned short int htobs( unsigned short int num );
unsigned short int btohs( unsigned short int num );
unsigned int htobl( unsigned int num );
unsigned int btohl( unsigned int num );
```

These functions convert 16- and 32-bit unsigned integers between the local computer's internal byte ordering (host order) and Bluetooth byte ordering. The function names describe the conversion. For example, `htobs` stands for Host to Bluetooth Short, indicating that it converts a short 16-bit unsigned integer from host order to Bluetooth order. One sure use is in specifying the port number in the `struct sockaddr_l2` structure, since 2 bytes are required to represent an L2CAP port number. Since RFCOMM port numbers can be represented with a single byte, byte ordering does not matter there. Other places the byte-order conversion functions may be used are in communicating with the Bluetooth microcontroller, performing low-level operations on transport protocol sockets, and implementing higher level Bluetooth profiles, such as the OBEX file transfer protocol.

Maximum Transmission Unit

Occasionally, an application may need to adjust the maximum transmission unit (MTU) for an L2CAP connection and set it to something other than the default of 672 bytes. This is done with the `struct l2cap_options` structure, and the `getsockopt` and `setsockopt` functions.

```
struct l2cap_options {
    uint16_t    omtu;
    uint16_t    imtu;
    uint16_t    flush_to;
    uint8_t     mode;
};
```

* Actually, only *most* of Bluetooth is little-endian. SDP is big-endian, while the rest of the core specification is little-endian. Yes, this is stupid.

```
int getsockopt( int sock, int level, int optname, void *optval,
                socklen_t *optlen );

int setsockopt( int sock, int level, int optname, void *optval,
                socklen_t optlen );
```

The `omtu` and `imtu` fields of the `struct l2cap_options` are used to specify the *outgoing MTU* and *incoming MTU*, respectively. The other two fields are currently unused and reserved for future use. To adjust the MTU for a connection, first use `getsockopt` to retrieve the existing L2CAP options for a connected socket. After modifying the options, `setsockopt` is used to apply the changes. For example, a function to do all of this might be as follows:

```
int set_l2cap_mtu( int sock, uint16_t mtu ) {
    struct l2cap_options opts;
    int optlen = sizeof(opts);
    int status = getsockopt( s, SOL_L2CAP, L2CAP_OPTIONS, &opts,
            &optlen );
    if( status == 0) {
        opts.omtu = opts.imtu = mtu;
        status = setsockopt( s, SOL_L2CAP, L2CAP_OPTIONS, &opts,
                optlen );
    }
    return status;
};
```

3.4 Service Discovery Protocol

The last step to building a robust Bluetooth application is making use of the Service Discovery Profile (SDP). The examples in this chapter so far have relied on hard-coded port numbers – not a good long-term solution. Additionally, client applications wishing to connect to a server have no way of programmatically finding out which nearby Bluetooth devices can provide the services they need. This section describes how to dynamically assign port numbers to server applications at runtime, and how to advertise and search for Bluetooth services using SDP.

Dynamically Assigned Port Numbers

The best way to get a dynamically assigned port number is actually to try binding to *every* possible port and stopping when `bind` doesn't fail. Aside from seeming a bit ugly, there's nothing wrong with this approach, and it will always work as long as a free port number is available. The following code snippet illustrates how to do this for RFCOMM sockets:

```
int sock, port, status;
struct sockaddr_rc to_bind;
sock = socket( AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM );
to_bind.rc_family = AF_BLUETOOTH;
to_bind.rc_bdaddr = *BDADDR_ANY;
for( port = 1; port <= 30; port++ ) {
    to_bind.rc_channel = port;
    status = bind(sock, (struct sockaddr *)&to_bind,
                  sizeof(to_bind));
    if( status == 0 ) break;
}
```

The process for L2CAP sockets is almost identical, but tries odd-numbered ports 4097–32,767 (0x1001–0x7FFF) instead of ports 1–30.

For Linux kernel versions 2.6.7 and beyond, it's possible to simply set the port number to 0 when filling out a socket addressing structure that gets passed to bind (a `struct sockaddr_rc` for RFCOMM, or a `struct sockaddr_l2` for L2CAP). During the call to `bind`, the kernel automatically chooses an available port number. To find out what port the kernel chose, use the `getsockname` function. This is probably a bit cleaner than exhaustively checking each port, but it's not guaranteed to be portable, especially on embedded and handheld devices that tend to use older kernels.

SDP Data Structures

Working with SDP in C can be a bit laborious because of the numerous data structures representing the data being passed back and forth between the application and an SDP server. Before getting into the details of how to register and search for services, here's a quick overview of the major data structures. If you're the type that likes to dive straight into examples, feel free to skip to the next section but remember this part for reference.

`sdp_record_t`: This represents a single service record advertised by an SDP server. It is a container data type used to consolidate all of the information in a service record. There are a number of functions used to manipulate the `sdp_record_t`, as we'll see later on.

`sdp_session_t`: This represents a connection to an SDP server, and is like a socket with SDP-specific functionality. Like the `sdp_record_t`, we won't have to deal directly with the data fields of this type, and will instead use helper functions introduced later on.

`uuid_t`: All Universally Unique Identifiers (UUIDs) are represented and manipulated as `uuid_t` data types. We'll often have to write code to fill them in, and there are three functions that we can use:

```
uuid_t* sdp_uuid128_create( uuid_t *uuid, const void *data );
uuid_t* sdp_uuid32_create( uuid_t *uuid, uint32_t data );
uuid_t* sdp_uuid16_create( uuid_t *uuid, uint16_t data );
```

Despite their names, all three functions create a 128-bit UUID. The difference is in whether the UUID is a reserved number or not. For unreserved UUIDs that a developer creates, use the `sdp_uuid128_create` function, which converts the 128 bits of memory starting at `data` into a `uuid_t`. For 32-bit and 16-bit reserved UUIDs, use the `sdp_uuid32_create` and `sdp_uuid16_create` functions, respectively.

`sdp_list_t`: Since SDP has very few fixed-length fields, pretty much everything is represented as a linked list of items, where each item can be of many different types, even other linked lists. `sdp_list_t` is a straightforward implementation of a linked list, with a number utility functions:

```
typedef struct _sdp_list sdp_list_t;
struct _sdp_list {
    sdp_list_t *next;
    void *data;
};

sdp_list_t *sdp_list_append( sdp_list_t *list, void *data );

void sdp_list_free( sdp_list_t *list, sdp_free_func_t f );
```

The `sdp_list_t` data type is used as both a pointer to an entire list and a pointer to an individual node in the list. It has two fields: `next`

points to the next node in the list, and `data` points to the data stored at a single node.

The `sdp_list_append` function is used for both adding nodes to a list and allocating new lists. To create a new linked list, set `list` to `NULL`, and the function allocates and returns a new list. To allocate and append a new node to the list, pass in the original list and the data element for the new node.

Once finished with a list, be sure to free the list memory with the `sdp_list_free` function. It is possible to have a utility function called on every data element in the list by passing a pointer to this function, when calling `sdp_list_free`. This utility function can be the standard `free` function or you can create a custom function to free the data elements. When the parameter `f` is set to `NULL`, then `sdp_list_free` does not modify or deallocate the data elements.

`sdp_profile_desc_t`: Whenever a service adheres to a Bluetooth profile, `sdp_profile_desc_t` is used to describe that profile:

```
typedef struct {
    uuid_t uuid;
    uint16_t version;
} sdp_profile_desc_t;
```

If a service advertises compliance with a Bluetooth profile, then it should advertise the UUID of that profile and the version number of the profile that it complies with.

`sdp_data_t`: An SDP service record consists of a list of entries, where each entry consists of an attribute/value pair. The `sdp_data_t` data type represents a *value* of that pair. Since the value can be of many different types (8-bit integer, 16-bit integer, text string, UUID, etc.) and can even be another `sdp_data_t`, this data type can be fairly complicated to deal with. It also has a few helper functions that will come in handy:

```
sdp_data_t * sdp_data_alloc( uint8_t dtd, const void *value );
sdp_attr_add(sdp_record_t *rec, uint16_t attr, sdp_data_t *data);
sdp_data_free( sdp_data_t *data );
```

The `sdp_data_alloc` function is used to allocate a new `sdp_data_t`. The `dtd` parameter specifies the type of data being allocated, and can

take on one of 32 different values. We'll only be using a few of them in our examples, but you can also check `bluetooth/sdp.h` for the full list.

3.4.1 Advertising a Service

Advertising a service can be broken up into two steps: creating the service record and registering it. Creating or building the service record can take up a fair amount of code, mostly because of the awkward way in which data structures are handled in C. The much simpler second step involves connecting to the local SDP server to actually register the service. The code in Example 3.6 shows how to do this, making use of the many helper functions to build the service record and register it with the local SDP server. The example advertises a service called "Roto-Rooter Data Router" running on RFCOMM port 11. The service claims to be in the Serial Port class of services, and also adheres to the Serial Port Profile. Additionally, it has a service ID of "00000000-0000-0000-00000000ABCD", which is poorly chosen but easy to read.

Example 3.6 Advertising a service

```
#include <unistd.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/sdp.h>
#include <bluetooth/sdp_lib.h>

sdp_session_t* register_service()
{
    uint32_t svc_uuid_int[] = { 0, 0, 0, 0xABCD };
    uint8_t rfcomm_channel = 11;
    const char *service_name = "Roto-Rooter Data Router";
    const char *svc_dsc = "An experimental plumbing router";
    const char *service_prov = "Roto-Rooter";

    uuid_t root_uuid, l2cap_uuid, rfcomm_uuid, svc_uuid,
           svc_class_uuid;
    sdp_list_t *l2cap_list = 0,
               *rfcomm_list = 0,
               *root_list = 0,
               *proto_list = 0,
               *access_proto_list = 0,
               *svc_class_list = 0,
               *profile_list = 0;
```

```
    sdp_data_t *channel = 0;
    sdp_profile_desc_t profile;
    sdp_record_t record = { 0 };
    sdp_session_t *session = 0;

    // set the general service ID
    sdp_uuid128_create( &svc_uuid, &svc_uuid_int );
    sdp_set_service_id( &record, svc_uuid );

    // set the service class
    sdp_uuid16_create(&svc_class_uuid, SERIAL_PORT_SVCLASS_ID);
    svc_class_list = sdp_list_append(0, &svc_class_uuid);
    sdp_set_service_classes(&record, svc_class_list);

    // set the Bluetooth profile information
    sdp_uuid16_create(&profile.uuid, SERIAL_PORT_PROFILE_ID);
    profile.version = 0x0100;
    profile_list = sdp_list_append(0, &profile);
    sdp_set_profile_descs(&record, profile_list);

    // make the service record publicly browsable
    sdp_uuid16_create(&root_uuid, PUBLIC_BROWSE_GROUP);
    root_list = sdp_list_append(0, &root_uuid);
    sdp_set_browse_groups( &record, root_list );

    // set l2cap information
    sdp_uuid16_create(&l2cap_uuid, L2CAP_UUID);
    l2cap_list = sdp_list_append( 0, &l2cap_uuid );
    proto_list = sdp_list_append( 0, l2cap_list );

    // register the RFCOMM channel for RFCOMM sockets
    sdp_uuid16_create(&rfcomm_uuid, RFCOMM_UUID);
    channel = sdp_data_alloc(SDP_UINT8, &rfcomm_channel);
    rfcomm_list = sdp_list_append( 0, &rfcomm_uuid );
    sdp_list_append( rfcomm_list, channel );
    sdp_list_append( proto_list, rfcomm_list );

    access_proto_list = sdp_list_append( 0, proto_list );
    sdp_set_access_protos( &record, access_proto_list );

    // set the name, provider, and description
    sdp_set_info_attr(&record, service_name, service_prov, svc_dsc);

    // connect to the local SDP server, register the service record,
```

```
    // and disconnect
    session = sdp_connect(BDADDR_ANY, BDADDR_LOCAL, SDP_RETRY_IF_BUSY);
    sdp_record_register(session, &record, 0);

    // cleanup
    sdp_data_free( channel );
    sdp_list_free( l2cap_list, 0 );
    sdp_list_free( rfcomm_list, 0 );
    sdp_list_free( root_list, 0 );
    sdp_list_free( access_proto_list, 0 );
    sdp_list_free( svc_class_list, 0 );
    sdp_list_free( profile_list, 0 );

    return session;
}

int main()
{
    sdp_session_t* session = register_service();
    sleep(5);
    sdp_close( session );
    return 0;
}
```

Before describing the individual data structures, help functions, and parameter descriptions, we overview the steps involved. The following table simply gathers together the comments in the code example, to make it easier to follow the explanation:

```
// set the general service ID
// set the service class
// set the Bluetooth profile information
// make the service record publicly browsable
// set l2cap information
// register the RFCOMM channel for RFCOMM sockets
// set the name, provider, and description
// connect to the local SDP server, register the service record,
// and disconnect
// cleanup
```

The code first declares a whole mess of local variables that will be used to store the different data elements of the service record, while the

real work starts by setting the Service ID using `sdp_uuid128_create` and `sdp_set_service_id`:

```
uuid_t* sdp_uuid128_create( uuid_t *uuid, const void *data );
void sdp_set_service_id( sdp_record_t *rec, uuid_t uuid );
```

The Service ID is specified as a full 128-bit number; there are no reserved Service IDs in Bluetooth, so any number is valid. It is natural to maintain the Service ID as an array of four 32-bit integers within the program, but this array must be converted to the `uuid_t` data type before being placed into the service record. The resulting `uuid_t` is passed to the function `sdp_set_service_id`, which fills in the appropriate field of the service record `rec`, which is also passed in as a parameter.

The next step is to create the Service Class List. For this example, the service advertises the reserved `SERIAL_PORT_CLASS` in its list of Service Classes. Since it's a reserved class, use `sdp_uuid16_create` to allocate the UUID. This is also the first place where we encounter the `sdp_list_t`, which is used to store the list of UUIDs. The function `sdp_set_service_classes` can then be used to apply the changes to the service record:

```
uuid_t* sdp_uuid16_create( uuid_t *uuid, uint16_t data );
sdp_list_t* sdp_list_append( sdp_list_t* list, void* data );
void sdp_set_service_classes( sdp_record_t* rec,
                              sdp_list_t* class_list );
```

The flow of data here is also straightforward. `sdp_uuid16_create` creates a Service Class ID, which is then passed to `sdp_list_append` to create a new linked list (as mentioned earlier, appending a data element to `NULL` creates a new list). This list is then passed to `sdp_set_service_classes`, which actually sets the Service Class List for the service record.

Creating and setting the Profile Descriptor List is similar, but instead of creating a list of UUIDs, create a list of `sdp_profile_desc_t` data structures (described earlier). The function `sdp_set_profile_descs` is then used to put or set this list in the service record:

```
void sdp_set_profile_descs( sdp_record_t* rec,
                            sdp_list_t* profile_list );
```

By now, the general idea of how to fill in a service record data structure is apparent. First, create an intermediate data structure that contains the information to set. Then, use one of the service record helper functions to

apply the changes to the master sdp_record_t data structure. Lather, rinse, and repeat. There are a few more of these helper functions in the example, and we'll quickly go over them here:

```
void sdp_set_browse_groups( sdp_record_t* rec,
                            sdp_list_t* browse_list );
void sdp_set_access_protos( sdp_record_t* rec,
                            sdp_list_t* proto_list );
void sdp_set_info_attr( sdp_record_t* rec, const char* name,
                        const char* provider,
                        const char* description );
```

The first of these is used to make the service record publicly browseable. By passing it a list that has a single UUID with value PUBLIC_BROWSE_GROUP, the application flags the service record for public browsing. Remote Bluetooth devices requesting a list of all available services (which we'll see how to do in the next section) will get this service record in the reply as a result of setting the public browse group.

The function sdp_set_access_protos sets in the service record the transport protocols that are to be advertised. This is also the place where the port number being used by the server application gets defined. This is a bit tricky because it actually takes a list of lists of lists (3 deep). The first inner list is supposed to represent a protocol stack, but you'll almost never have more than one of these. Within each protocol stack list, you'll have one list for each transport protocol used by the service. Since RFCOMM is built on top of L2CAP, all RFCOMM applications always have at least an L2CAP list and an RFCOMM list. The third inner list contains the details for the protocol list, and usually has one or two items. The first item should be a UUID identifying the protocol. If the second item is present, it should be a sdp_data_t specifying the port number used by the service. The dtd field of the sdp_data_t should be SDP_UINT8 for RFCOMM ports and SDP_UINT16 for L2CAP ports. Confusing, isn't it? The example code should actually work in most cases with minor modifications, so don't get too hung up on figuring it all out.

The sdp_set_info_attr function can be used to set three fields all at once, all of them text fields. name should be the name of the service provided, provider is supposed to be the provider of the service, and description describes the service. All three of these fields are meant to be human readable and not

interpreted or specially parsed by Bluetooth programs, so they can really be whatever you want them to be. Setting any of the three parameters to `NULL` causes it to not be included in the service record.

Finally, we're done constructing the service record! Congratulate yourself, and breathe a sigh of relief. The rest of advertising a service is easy, and we only need three more functions:

```
sdp_session_t *sdp_connect( const bdaddr_t *src,
                            const bdaddr_t *dst, uint32_t flags );
int sdp_record_register( sdp_session_t *session, sdp_record_t *rec,
                         uint8_t flags );
int sdp_close( sdp_session_t *session );
```

First, use the `sdp_connect` function to connect to the SDP server running on the local machine. The first parameter, `src`, should always be `BDADDR_ANY`; the second parameter should always be `BDADDR_LOCAL`; and the third parameter should always be `0`. Later on, we'll use different values for these parameters, but they should always be the same when advertising a service.

The function `sdp_connect` returns a pointer to a newly allocated `sdp_session_t`, which represents a connection to the local SDP server. This pointer then gets passed to `sdp_record_register` along with the service record that we so carefully constructed. This function finishes the registration process, and the program is now free to go on with the rest of its tasks. The service record will stay registered and advertised until the program exits or closes the connection to the local SDP server by calling `sdp_close`.

Searching and Browsing for a Service

The process of searching for services involves two steps: detecting all nearby devices with a device inquiry, and connecting to each of those devices in turn to search for the desired service.* The first step, detecting all nearby devices, was covered in Section 3.1, so we'll just skip that and move right on to the second step.

* You might say, "Well, why isn't there a way to broadcast service searches?" and to that, we would say, "Good question!" Despite the piconet abilities of Bluetooth, there has been no way for a device to (metaphorically) shout out, "Does anyone have a printer?" Instead, a client application has to do the equivalent of walking up to each nearby device and saying, "Excuse me, can I have a minute? Yes, do you have a printer available? No? Okay, sorry to bother." With the introduction of Extended Inquiry Response in Bluetooth 2.1, this may change.

Once connected to the SDP server on a remote Bluetooth device, a client can search on a specific UUID. The remote device should then return a list of all services that have that UUID anywhere in the service record. The UUID could match the record's Service ID, one of its Service Classes, or even the transport protocol used by the service. Example 3.7 shows how to search a single device to see if it has an RFCOMM service with UUID `00000000-0000-0000-0000-00000000ABCD`. An explanation follows:

> *Note:* Browsing, or requesting a list of all services a device has to offer, is actually a special case of searching. All publicly available services on a device will have the reserved UUID `PUBLIC_BROWSE_GROUP` as an attribute, so searching for that UUID is equivalent to asking for all public services on a device.

Example 3.7 Searching for a service using SDP.

```
#include <stdio.h>
#include <stdlib.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/sdp.h>
#include <bluetooth/sdp_lib.h>

int main(int argc, char **argv)
{
    uint32_t svc_uuid_int[] = { 0, 0, 0, 0xABCD };

    int status;
    bdaddr_t target;
    uuid_t svc_uuid;
    sdp_list_t *response_list, *search_list, *attrid_list;
    sdp_session_t *session = 0;
    uint32_t range = 0x0000ffff;
    uint8_t port = 0;

    if(argc < 2)
    {
```

```
        fprintf(stderr, "usage: %s <bt_addr>\n", argv[0]);
        exit(2);
    }
    str2ba( argv[1], &target );

    // connect to the SDP server running on the remote machine
    session = sdp_connect( BDADDR_ANY, &target, 0 );

    sdp_uuid128_create( &svc_uuid, &svc_uuid_int );
    search_list = sdp_list_append( 0, &svc_uuid );
    attrid_list = sdp_list_append( 0, &range );

    // get a list of service records that have UUID 0xabcd
    response_list = NULL;
    status = sdp_service_search_attr_req( session, search_list,
            SDP_ATTR_REQ_RANGE, attrid_list, &response_list);

    if( status == 0 ) {
        sdp_list_t *proto_list = NULL;
        sdp_list_t *r = response_list;

        // go through each of the service records
        for (; r; r = r->next ) {
            sdp_record_t *rec = (sdp_record_t*) r->data;

            // get a list of the protocol sequences
            if( sdp_get_access_protos( rec, &proto_list ) == 0 ) {

                // get the RFCOMM port number
                port = sdp_get_proto_port( proto_list, RFCOMM_UUID );
                sdp_list_free( proto_list, 0 );
            }
            sdp_record_free( rec );
        }
    }
    sdp_list_free( response_list, 0 );
    sdp_list_free( search_list, 0 );
```

```
    sdp_list_free( attrid_list, 0 );
    sdp_close( session );

    if( port != 0 ) {
        printf("found service running on RFCOMM port %d\n", port);
    }

    return 0;
}
```

The code in the example starts off by connecting to a specific Bluetooth device (the one with address `01:23:45:67:89:AB`) using the `sdp_connect` function that we saw in the previous section.

```
sdp_session_t *sdp_connect( const bdaddr_t *src,
                            const bdaddr_t *dst, uint32_t flags );
```

This time around, the `dst` parameter to `sdp_connect` is set to the address of the remote Bluetooth device. If your application needs to use a specific local Bluetooth adapter to conduct the search, then pass its address in as the `src` parameter, but otherwise just leave it set to `BDADDR_ANY`. Don't worry about the `flags` parameter, it doesn't really do much and so just leave it at 0. If the system isn't able to connect to the remote SDP server, then `sdp_connect` returns `NULL` instead of a valid pointer.

Once connected, the client program prepares to send its search query by creating two lists. The first list contains the UUIDs that the client is searching for. In this example, the client uses `sdp_uuid128_create` to make a single UUID. When searching for a standard reserved UUID, the usual case, use the function `sdp_uuid16_create` or `sdp_uuid32_create` (described earlier in this chapter). Multiple UUIDs can be searched at the same time, just append more of them to the list, and only service records matching every UUID will be returned.

You can use the second list to control exactly what attribute/value pairs of matching service records that an SDP server returns during a search, but usually we just want the SDP server to send us everything it has for matching service records. To do this, just populate it with a single 32-bit integer with value `0xFFFF`.

Search terms in hand, the client program sends the search query using the sdp_service_search_attr_req function:

```
int sdp_service_search_attr_req( sdp_session_t* session,
      const sdp_list_t* uuid_list, sdp_attrreq_type_t reqtype,
      const sdp_list_t* attrid_list, sdp_list_t **response_list );
```

The first parameter to this function is a pointer to the sdp_session_t created above. The second parameter, uuid_list, is the list of UUIDs just created, and attrid_list is the list containing the single 32-bit integer also just created. Leave reqtype set to SDP_ATTR_REQ_RANGE, and pass the address of a NULL pointer in as a response_list. This last one is an output parameter, which will point to a newly allocated sdp_list_t when the function completes. sdp_service_search_attr_req returns 0 when the search completes successfully (which doesn't necessarily mean that it got any results, just that it communicated with the SDP server successfully), and −1 on failure.

After a successful search, the client program will then have a linked list of service records to parse through. These are the same sdp_record_t data structures that were created by the server application to advertise its service as described above. This time, however, the program is on the receiving side and must slog through them to find what it needs.

> *Note:* The last node of an sdp_list_t linked list has NULL as its next field. To iterate through a list, a program can traverse the next links until it reaches NULL.

Extracting information from an sdp_record_t data structure involves a number of helper functions. Typically, the data structure is not accessed directly, but rather through the functions of the form *sdp_get_ATTR*, where *ATTR* will be some attribute, such as sdp_get_service_classes.

Since a client program is primarily interested in figuring out how to connect to the service being advertised by the SDP server, it should focus its attention on the the list of transport protocols in the service record. To get to this list, use the functions sdp_get_access_protos and sdp_get_proto_port:

```
int sdp_get_access_protos( const sdp_record_t *rec,
                           sdp_list_t **proto_list );
int sdp_get_proto_port( const sdp_list_t *proto_list,
                        int proto_uuid );
```

To determine the port on which to access a service, pass a `sdp_record_t` from the search results into `sdp_get_access_protos` along with the address of a `NULL` pointer. The `proto_list` is an output parameter, and will point to a newly allocated `sdp_list_t` when the function completes successfully. This list represents all protocols and ports advertised in the service record. `sdp_get_proto_port` can then be used to extract the port number. Pass it the protocol list and either `RFCOMM_UUID` (for RFCOMM services) or `L2CAP_UUID` (for L2CAP services). The function returns the port number used by the service, or 0 if it couldn't find one.

Figuring out the port number that a service is running on is usually the most important part of searching with SDP, so in that respect we're all done. Other attributes of an advertised service record can also be useful, however, and the following helper functions can be used to access them:

Service ID

```
int sdp_get_service_id(const sdp_record_t *rec, uuid_t *uuid);
```

The service ID will be stored in output parameter `uuid`, which should point to a valid `uuid_t`.

Service Class List

```
int sdp_get_service_classes(const sdp_record_t *rec,
                            sdp_list_t **service_class_list);
```

`service_class_list` should be the address of a `NULL` pointer, which will be changed to point to a newly allocated `sdp_list_t`. This will be a list of `uuid_t` data structures, each of which is the UUID of a service class of the service record.

Profile Descriptor List

```
int sdp_get_profile_descs(const sdp_record_t *rec,
                          sdp_list_t **profile_descriptor_list);
```

`profile_descriptor_list` should be the address of a `NULL` pointer, which will be changed to point to a newly allocated `sdp_list_t`. This will be a list of `sdp_profile_desc_t` data structures, each of which describes a Bluetooth Profile that the service adheres to.

Service Name, Service Provider, and Service Description

```
int sdp_get_service_name(const sdp_record_t *rec, char *buf,
                         int len);
```

```
int sdp_get_service_desc(const sdp_record_t *rec, char *buf,
                         int len);
int sdp_get_provider_name(const sdp_record_t *rec, char *buf,
                          int len);
```

All three of these functions copy a text string into the output parameter `buf`. The `len` is a size limit, but it's not quite what you might expect. If the actual attribute is longer than `len` bytes, then all three functions will fail and return −1. Otherwise, the full attribute text is copied into the buffer. It's probably best to just set this to a large, healthy number.

3.5 Advanced BlueZ Programming

At this point, the essentials for creating a Bluetooth application have been covered, and the reader should be well equipped to tackle a problem requiring simple Bluetooth techniques. The rest of the chapter focuses on aspects of Bluetooth that may be necessary for larger applications. Section 3.5.1 describes asynchronous socket programming, a very valuable tool given the fact that Bluetooth devices can take several seconds to respond. The remaining sections may be relevant to applications that require more specialized Bluetooth techniques. Section 3.5.2 explains how to access the HCI transport layer, with a demonstration on adjusting the L2CAP maximum packet size. Section 3.5.3 shows how to use the best-effort transmission protocols. Finally, Section 3.5.4 provides a brief introduction to SCO sockets for voice-quality audio transmission.

3.5.1 Asynchronous Socket Programming with `select`

Application programs nearly always want to do something else or at least be able to respond to user input while waiting for nearby Bluetooth devices to respond. The code in Example 3.8 demonstrates the use of nonblocking sockets to establish an outgoing connection. The counterpart, using a nonblocking socket to accept an incoming connection, is similar but is not shown.

Example 3.8 rfcomm-client-select.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
```

```
#include <sys/socket.h>
#include <sys/select.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>

int main(int argc, char **argv)
{
    struct sockaddr_rc addr = { 0 };
    int s, status;
    char dest[18] = "01:23:45:67:89:AB";
    fd_set readfds, writefds;
    int maxfd, sock_flags;

    // allocate a socket
    s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

    // set the connection parameters (who to connect to)
    addr.rc_family = AF_BLUETOOTH;
    addr.rc_channel = 1;
    str2ba( dest, &addr.rc_bdaddr );

    // put socket in non-blocking mode
    sock_flags = fcntl( s, F_GETFL, 0 );
    fcntl( s, F_SETFL, sock_flags | O_NONBLOCK );

    // initiate connection attempt
    status = connect(s, (struct sockaddr *)&addr, sizeof(addr));
    if( 0 != status && errno != EAGAIN ) {
        perror("connect");
        return 1;
    }

    // wait for connection to complete or fail
    FD_ZERO(&readfds);
    FD_ZERO(&writefds);
    FD_SET(s, &writefds);
    maxfd = s;
```

```
    status = select(maxfd + 1, &readfds, &writefds, NULL, NULL);
    if( status > 0 && FD_ISSET( s, &writefds ) ) {
        status = send(s, "hello!", 6, 0);
    }
    if( status < 0 ) perror("operation failed");

    close(s);
    return 0;
}
```

Before going through the code example, note that most of the constants and header files for nonblocking sockets in Bluez are found in the `unistd.h`, `fcntl.h`, and `sys/select.h` header files; they should always be included in your program.

The first step in asynchronous socket programming is to place each socket in nonblocking mode:

```
sock_flags = fcntl( s, F_GETFL, 0 );
fcntl( s, F_SETFL, sock_flags | O_NONBLOCK );
```

Here, we see two calls to the standard Linux function `fcntl`. The first invocation retrieves the existing flags associated with the socket. Each bit of `sock_flags` corresponds to a single option, one of which is the nonblocking option. The second invocation sets the options while enabling the nonblocking bit. This must be done for each socket that is to be used asynchronously, including sockets that have been created by a call to `accept`; a nonblocking server socket does not transfer its nonblocking properties to newly created client sockets.

Once a socket is in nonblocking mode, calls to normally blocking functions will return a -1 and set `errno` to `EAGAIN`. It is important that applications handle these cases and not treat them as true errors.

Next, each nonblocking socket should be placed into one or both of the read and write sets. BlueZ does not use the exception set.

```
fd_set readfds, writefds;

FD_ZERO(&readfds);
FD_ZERO(&writefds);
```

```
FD_SET(s, &writefds);

maxfd = s;
```

An `fd_set` is a data type that represents a set of file descriptors. (A socket is a special type of file descriptor.) The `FD_ZERO` macro is used to initialize and clear an `fd_set`, and the `FD_SET` macro adds a file descriptor to the set if it is not already present. Additionally, `maxfd` is used to keep track of the biggest file descriptor, which is needed during a call to `select`.

A timeout, indicating the maximum amount of time that `select` should spend waiting for an event, can be specified if needed:

```
struct timeval timeout;
timeout.tv_sec = 1;
timeout.tv_usec = 500000;
```

The `tv_sec` field denotes seconds, and `tv_usec` denotes microseconds. This specific example indicates 1.5 s.

Once all of this is in place, a call to `select` can be made:

```
status = select(maxfd + 1, &readfds, &writefds, NULL, &timeout);
if( status > 0 && FD_ISSET( s, &writefds ) ) {
    status = send(s, "hello!", 6, 0);
}
```

The first parameter to `select` should be the largest numbered file descriptor in either the read or write sets, plus 1. Next, the three sets are passed in. The value `NULL` can be used in place of a set if that set is not needed. Here, `NULL` is passed instead of an exception set. Finally, a timeout can be specified, or `NULL` to instruct `select` to return only when an event has occurred. This is different from passing a timeout with both fields set to 0, in which case `select` always returns immediately.

When `select` returns, the three sets are modified to contain only file descriptors for which events have occurred. The return value of `select` indicates the total number of these file descriptors, a negative number on error, or 0 if the timeout passed without any events of interest. When an event does occur, the `FD_ISSET` macro can then be used to check a socket and see if an event should be processed.

Since `select` modifies the file descriptor sets, they should be reinitialized and populated before each invocation. `timeout` should also be reinitialized, as

the GNU/Linux implementation of `select` modifies it to contain the amount of time remaining.

3.5.2 HCI Sockets

In addition to the L2CAP and RFCOMM sockets described in this chapter, BlueZ provides a number of other socket types. The most useful of these is the HCI socket, which provides a direct connection to the microcontroller on the local Bluetooth adapter. This socket type, introduced in Section 3.1, can be used to issue arbitrary commands to the Bluetooth adapter. Programmers requiring precise control over the Bluetooth controller to perform tasks such as asynchronous device discovery or reading signal strength information should use HCI sockets.

The simplest way to create an HCI socket is to use `hci_open_dev` in conjunction with `hci_get_route`, both of which were described earlier in the chapter.

The semantics of using an HCI socket are fairly simple. The host computer can send commands to the microcontroller by writing to the socket, and the microcontroller generates events to indicate command responses and other status changes. These events can then be read from the socket.

A command consists of three parts – an Opcode Group Field that specifies the general category of the command, an Opcode Command Field that specifies the actual command, and a series of command-specific parameters. Events have two parts – an event code that specifies the type of event and a series of parameters that depend on the event code. If you wanted to get down and dirty with the Bluetooth specification, you could programmatically pack and unpack commands and events, and use `send` and `recv` for all your communication. For the less adventurous, there are also the `hci_send_cmd` and `hci_send_req` functions:

```
int hci_send_cmd(int sock, uint16_t ogf, uint16_t ocf, uint8_t clen,
                 void *cparam);

struct hci_request {
    uint16_t ogf;
    uint16_t ocf;
    int      event;
    void     *cparam;
    int      clen;
    void     *rparam;
    int      rlen;
};
int hci_send_req(int sock, struct hci_request *req, int timeout);
```

The `hci_send_cmd` function sends a single command to the Bluetooth microcontroller, and `hci_send_req` both sends a command and waits for a response. There are five parameters of `hci_send_cmd`: `sock` is an open HCI socket, `ogf` is the Opcode Group Field, `ocf` is the Opcode Command Field, and `clen` specifies the length in bytes of the command parameters `cparam`.

The parameters to function `hci_send_request` are mostly embedded in the `struct hci_request` data structure that is passed in to the function. `ogf`, `ocf`, `cparam`, and `clen` take on the same meaning as before. `event` specifies the event code of the event to wait for, and `rlen` should be the size of the `rparam` buffer, which will be filled in with the event parameters. When using `hci_send_request`, `timeout` specifies the maximum number of milliseconds to wait for the command to complete before timing out. To never timeout, use 0 instead. Example 3.9 in the next section demonstrates how to use `hci_send_request`.

BlueZ also provides a host of convenience functions that are more or less short wrappers around the `hci_send_cmd` and `hci_send_request` functions with hard-coded parameters. For example, to change the user-friendly name of a local Bluetooth adapter, you could use `hci_write_local_name`. All of these functions are prefixed with *hci_*, and can be found in the `hci_lib.h` header file.

We're not going to bother listing all the gory details of the commands that can be sent to the Bluetooth microcontroller, nor will we list all the HCI functions available; we do not consider them essential. For programming at such a low level, you'd be best served by going straight to the Bluetooth Core Specification, which has a well-written section on using HCI and all the commands, events, and parameters available. If you know the command or event, browsing through the `hci.h` and `hci_lib.h` header files should make it obvious which functions and data structures to use.

3.5.3 L2CAP Best-Effort Transmission

The semantics of an L2CAP connection can be changed from reliable to a best-effort transmission policy, by adjusting the *flush timeout* of a connection. By default, L2CAP provides reliable transmission guarantees using a transmit/acknowledge scheme. The sender of a packet always waits for an acknowledgment from the receiver before sending the next packet, so that packets will either be reliably delivered in order or the connection fails. In other words, with reliable transmission, there is no timeout value. A best-effort transmission policy specifies the maximum time to wait for a packet

acknowledgment before moving on to the next packet. The details of changing this value are a little complicated, however.

Multiple L2CAP and RFCOMM connections between two devices are actually logical connections multiplexed on a single, lower level ACL connection. There is only one flush timeout for each ACL connection, and adjusting it affects *all* L2CAP and RFCOMM connections between the two devices. Most existing Bluetooth profiles and applications do not make use of this option, but it is possible to do so using HCI sockets in BlueZ. Example 3.9 demonstrates how to do this with `hci_send_request`.

A handle to the underlying connection is first needed to make this change, but the only way to obtain a handle to the underlying connection is to query the microcontroller on the local Bluetooth adapter. Once the connection handle has been determined, a command can be issued to the microcontroller instructing it to make the appropriate adjustments.

Example 3.9 set-flush-to.c

```
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>

int set_flush_timeout(bdaddr_t *ba, int timeout)
{
    int err = 0, dd;
    struct hci_conn_info_req *cr = 0;
    struct hci_request rq = { 0 };

    struct {
        uint16_t handle;
        uint16_t flush_timeout;
    } cmd_param;

    struct {
        uint8_t  status;
        uint16_t handle;
    } cmd_response;
```

```
    // find the connection handle to the specified bluetooth device
    cr = (struct hci_conn_info_req*) malloc(
            sizeof(struct hci_conn_info_req) +
            sizeof(struct hci_conn_info));
    bacpy( &cr->bdaddr, ba );
    cr->type = ACL_LINK;
    dd = hci_open_dev( hci_get_route( &cr->bdaddr ) );
    if( dd < 0 ) {
        err = dd;
        goto cleanup;
    }
    err = ioctl(dd, HCIGETCONNINFO, (unsigned long) cr );
    if( err ) goto cleanup;

    // build a command packet to send to the bluetooth microcontroller
    cmd_param.handle = cr->conn_info->handle;
    cmd_param.flush_timeout = htobs(timeout);
    rq.ogf = OGF_HOST_CTL;
    rq.ocf = 0x28;
    rq.cparam = &cmd_param;
    rq.clen = sizeof(cmd_param);
    rq.rparam = &cmd_response;
    rq.rlen = sizeof(cmd_response);
    rq.event = EVT_CMD_COMPLETE;

    // send the command and wait for the response
    err = hci_send_req( dd, &rq, 0 );
    if( err ) goto cleanup;

    if( cmd_response.status ) {
        err = -1;
        errno = bt_error(cmd_response.status);
    }

cleanup:
    free(cr);
    if( dd >= 0) close(dd);
    return err;
}

int main(int argc, char **argv)
{
    bdaddr_t target;
    int timeout;
```

```
    if( argc < 3 ) {
        fprintf(stderr, "usage: set-flush-to <addr> <timeout>\n");
        exit(2);
    }

    str2ba( argv[1], &target );
    timeout = atoi( argv[2] );
    return set_flush_timeout( &target, timeout );
}
```

Setting the flush timeout for a connection involves two steps: First, the program uses `ioctl` to retrieve a connection handle to the lower level Bluetooth connection with another device. Second, it uses `hci_send_request` (described in the HCI sockets section) to send the adjustment command to the local Bluetooth microcontroller.

On success, the packet timeout for the low-level connection to the specified device is set to `timeout * 0.625` ms. A timeout of 0 is used to indicate infinite, and can be used to revert to a reliable connection. The bulk of this example is comprised of code to construct the command packets and response packets used in communicating with the Bluetooth controller. The Bluetooth Specification defines the structure of these packets and the magic number `0x28`.

3.5.4 SCO Audio Sockets

A major use of Bluetooth communication is for transmitting voice-quality audio between a wireless headset and either a telephone or a computer. The SCO protocol is used in these cases to transfer the actual audio. To use SCO sockets, include `bluetooth/sco.h`. SCO sockets can be allocated by passing the socket type `SOCK_SEQPACKET` and protocol `BTPROTO_SCO` to the `socket` function:

```
int sco_sock = socket( AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_SCO );
```

The `struct sockaddr_sco` addressing structure is used with the `bind`, `connect`, and `accept` functions:

```
struct sockaddr_sco {
    sa_family_t sco_family;
    bdaddr_t    sco_bdaddr;
};
```

As with the RFCOMM and L2CAP addressing structures, the `sco_family` field is always set to `AF_BLUETOOTH`. The `sco_bdaddr` field specifies the local adapter to use, the remote device to connect to, or the address of a newly connected device for the `bind`, `connect`, and `accept` functions, respectively.

Aside from these differences, the semantics of using SCO sockets are the same as for all the other socket types, with a few details to watch out for. The two most important of these are the format of the audio packets being sent, and the MTU of these packets.

Before establishing an SCO connection, the `hci_read_voice_setting` and `hci_write_voice_setting` functions should be used to check and set the audio format used for future SCO connections:

```
int hci_read_voice_setting(int dd, uint16_t *vs, int to);
int hci_write_voice_setting(int dd, uint16_t vs, int to);
```

The parameter `dd` should be an HCI socket, `vs` is used to store the audio format, and `to` is a timeout (milliseconds) to wait for each function to complete. The exact format of the voice setting is described in Section 6.12 of the Bluetooth 2.0 core specification, and it is important that *both* devices use the same setting. Many Bluetooth chips use a value of `0x0060` as a default, as this is the voice setting specified for the Bluetooth Headset Profile. For this particular setting, data sent through the socket with the `send` or `write` functions should correspond to signed 16-bit mono samples at 8 kHz. The packets will be automatically passed through a continuously variable slope delta codec during the transmission process.

Once a connection is established, the application should check the maximum packet size and adjust its transmitted packets accordingly. This is done using the `getsockopt` function:

```
struct sco_options so;
int so_size = sizeof( so );
getsockopt( connected_sco_socket, SOL_SCO, SCO_OPTIONS, &so,
            &so_size );
uint16_t maximum_packet_size = so.mtu;
```

Readers who have carefully studied Chapter 2 may notice that the way to determine the MTU is different here than it is in PyBluez (which used an HCI socket and `hci_send_req`). Both ways actually return the same information, and it is also possible to use `hci_send_req` to determine the SCO MTU here, but `getsockopt` is simpler and cleaner in this case.

3.6 Summary

In this chapter, we have seen how to apply the concepts and techniques introduced in Chapter 1 to creating Bluetooth applications with the BlueZ development libraries. Bluetooth application development in C can be verbose and tedious, but ultimately allows the developer great control over the behavior and performance of the application.

Device discovery
Name lookup
	`WSALookupServiceBegin( ... ,`
	`  flags = LUP_CONTAINERS | ...,  );`
	`WSALookupServiceNext( ... );`
	`WSALookupServiceEnd( ... );`

SDP connect
SDP search
	`WSALookupServiceBegin( ... );`
	`WSALookupServiceNext( ... );`
	`WSALookupServiceEnd( ... );`

Establish an outgoing connection	`int s = socket(AF_BTH, SOCK_STREAM,`
	`  BTHPROTO_RFCOMM);`
	`connect(s, ...);`

Establish an incoming connection	`int s = socket(AF_BTH, SOCK_STREAM,`
	`  BTHPROTO_RFCOMM);`
	`bind(s, ...);`
	`listen(s, backlog);`
	`accept(s, ...);`

Advertise an SDP service	`WSAQUERYSET sinfo = { ... };`
	`WSASetService(&sinfo,`
	`  RNRSERVICE_REGISTER, 0);`

Transfer data	`send(s, data, datalen, flags);` `recv(s, buffer, buflen, flags);`
Disconnect	`close(s);`

Microsoft Bluetooth API Quick Reference

Chapter 4

C Programming with Microsoft Windows XP

The essentials of Bluetooth programming under Microsoft Windows XP follow the general trend, although there are complications. With the introduction of Service Pack 1, Microsoft began natively supporting Bluetooth in Windows XP. Many devices are now natively supported, and can be programmed using the Platform SDK, which is available for download from the Microsoft web site. Prior to this native support, many Bluetooth stacks were developed for the Windows environment. One of the more popular Bluetooth development environments was, and still is, the Widcomm Bluetooth SDK.* Table 4.1 illustrates the transport protocols supported by the Microsoft and Widcomm Bluetooth development kits. In addition to the protocols listed, the Widcomm development kit also provides support for OBEX (object exchange), printing, DUN, and A2DP.

By introducing their own Bluetooth drivers, libraries, and APIs, Microsoft also caused a fair amount of confusion for many Bluetooth users. Users often install a third-party Bluetooth stack and API that come with Bluetooth devices that attach to the PC. All of a sudden, two completely separate sets of software applications would be vying for control of a single Bluetooth device. The user may have to disable one of the competing software stacks so that the other

* The Broadcom Corporation acquired Widcomm in April 2004, so the Widcomm SDK is often referred to as the Broadcom SDK.

Table 4.1 Transport protocols supported by the Microsoft and Widcomm stacks.

Microsoft	RFCOMM	L2CAP	SCO	HCI
Widcomm / Broadcom	RFCOMM	L2CAP	SCO	HCI

could have unfettered access to the local Bluetooth devices. It is not hard to imagine the resulting confusion.

The lack of standard Bluetooth stack interfaces and API means that Bluetooth software developers targeting the Windows platform now need to choose to either write their software to link against a single API or to have it linked against multiple APIs. The latter approach is naturally more reliable and increases the chances that their software "Just Works," but is significantly more time consuming and expensive. Developers without the resources or patience to support multiple Bluetooth stacks would then need to choose an API to use.

So which API is the better way to go? Equivalently, where should you start? Well, it depends. First, you may not actually have a choice. Older operating systems, such as Windows 98, ME, and 2000, still require a third-party Bluetooth stack. Additionally, the Microsoft Bluetooth stack provides support only for the RFCOMM transport protocol. Audio applications that use SCO connections will need to use a different stack. Applications needing the OBEX File Transfer Profile (FTP) or the Object Push Profile may find it easier to use a third-party API that already provides abstractions and API support for those profiles.

Although the Microsoft Bluetooth stack is limited, it does have the advantage of coming standard with Windows XP SP2. This chapter shows how to program with the Microsoft Bluetooth API. It does not cover how to set up a development environment, since doing so is already described in many other places. We assume that the interested reader already has a working installation of Microsoft Visual C++ and the Microsoft Windows XP Platform SDK.*

* http://www.microsoft.com/msdownload/platformsdk

> *Tip:* Students and others without a full copy of Microsoft Visual Studio can create Bluetooth applications using Microsoft Visual Studio Express, which is freely distributed on the Microsoft Web site at `http://msdn.microsoft.com/vstudio/express`. Although Visual Studio Express doesn't have all the features available in Visual Studio (most notably, the Microsoft Foundation Classes), it is still quite versatile.

The chapter first addresses some Microsoft-specific issues. The bulk of the chapter explains how to find nearby Bluetooth devices, as well as how to advertise and search for specific services. Most of the functionality is embedded in the data structures passed to a handful of functions.

4.1 Preliminaries

The Microsoft Bluetooth API extends the Windows Sockets 2 API, which is usually used for Internet programming. Using it for Bluetooth programming is identical in nature to the methods we've seen in previous chapters, with the main differences lying in the layout of the data structures and the syntactic details.

Before getting into the specifics, we first show how to initialize sockets and give an overview of the basic data structures. Subsequent sections discuss the specifics of these structures.

4.1.1 Header Files and Linked Libraries

Three header files are needed to enable use of the Bluetooth extensions. The following must be included and in this order: `winsock2.h`, `ws2bth.h`, and `BluetoothAPIs.h`. The program should be linked against `ws2_32.lib` and `irprops.lib`. These files are all included with the Windows Platform SDK.

4.1.2 Initializing the Windows Sockets API

As with all programs using the Windows Sockets 2 API, some initialization code is required before the use of any socket functions. Programs should

define and invoke something similar to the following function before using any Bluetooth resources:

```
#include <winsock2.h>
#include <ws2bth.h>

void initialize_windows_sockets() {
    WSADATA wsaData;
    WORD wVersionRequested = MAKEWORD( 2, 0 );
    if( WSAStartup( wVersionRequested, &wsaData ) != NO_ERROR ) {
        fprintf(stderr, "Error initializing window sockets!\n");
        ExitProcess(2);
    }
}
```

This needs to be invoked only once per process. Once a program is finished using Bluetooth resources, it should call `WSACleanup` to release resources used by the Windows Sockets 2 API:

```
int WSACleanup(void);
```

4.1.3 Error Checking

All of the functions introduced in this section, with the exception of `WSAStartup`, return `NO_ERROR` on success and `SOCKET_ERROR` on failure. If something goes wrong, the function `WSAGetLastError` can be used to retrieve an error code and `FormatMessage` to convert the error code to a readable string. Full details on the error codes and these two functions can be found in the Windows SDK documentation, but you may find it helpful to define a function like the following when debugging your application:

```
void PrintLastError()
{
    LPVOID lpMsgBuf;
    FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
                   FORMAT_MESSAGE_FROM_SYSTEM |
                   FORMAT_MESSAGE_IGNORE_INSERTS,
            NULL, GetLastError(), 0, (LPTSTR) &lpMsgBuf, 0, NULL );
    fprintf(stderr, "%s\n", lpMsgBuf); // or MessageBox() for GUIs
    free(lpMsgBuf);
}
```

4.1.4 Data Structures

Like other C-based APIs, the Microsoft Bluetooth API has its share of data structures to learn. The following five data structures are the most relevant and will be the subject of the coming subsections. We suggest just skimming them at first, without trying very hard to remember the details. As each data structure is introduced we'll refer back to this figure:

```
typedef ULONGLONG BTH_ADDR;       // represents a Bluetooth address

typedef struct _SOCKADDR_BTH {    // socket addressing structure
    USHORT addressFamily;         // used for establishing connections
    BTH_ADDR btAddr;              // and retrieving connection details
    GUID serviceClassId;
    ULONG port;
} SOCKADDR_BTH;

typedef struct _SOCKET_ADDRESS {  // container for SOCKADDR_BTH
    LPSOCKADDR lpSockaddr;        // used for device discovery and SDP
    INT iSockaddrLength;
} SOCKET_ADDRESS;

typedef struct _CSADDR_INFO {     // container for SOCKET_ADDRESS
    SOCKET_ADDRESS LocalAddr;     // used for device discovery and SDP
    SOCKET_ADDRESS RemoteAddr;
    INT iSocketType;
    INT iProtocol;
} CSADDR_INFO;
typedef struct _GUID {            // represents a UUID, used with SDP
    DWORD Data1;
    WORD Data2;
    WORD Data3;
    BYTE Data4[8];
} GUID;

typedef struct _WSAQuerySet {     // input/output structure for device
    DWORD dwSize;                 // discovery and SDP
    LPTSTR lpszServiceInstanceName;
    LPGUID lpServiceClassId;
    LPWSAVERSION lpVersion;
    LPTSTR lpszComment;
    DWORD dwNameSpace;
    LPGUID lpNSProviderId;
    LPTSTR lpszContext;
    DWORD dwNumberOfProtocols;
    LPAFPROTOCOLS lpafpProtocols;
```

```
    LPTSTR lpszQueryString;
    DWORD dwNumberOfCsAddrs;
    LPCSADDR_INFO lpcsaBuffer;
    DWORD dwOutputFlags;
    LPBLOB lpBlob;
} WSAQUERYSET;
```

4.2 Choosing a Remote Device

As usual, we begin with a description of detecting nearby Bluetooth devices. Example 4.1 demonstrates this, and an explanation follows.

Example 4.1 Device discovery with the Microsoft Bluetooth API.

```
#include "stdafx.h"
#include <winsock2.h>
#include <ws2bth.h>

int _tmain(int argc, _TCHAR* argv[])
{
    // setup windows sockets
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested = MAKEWORD( 2, 0 );
    if( WSAStartup( wVersionRequested, &wsaData ) != 0 ) {
        fprintf(stderr, "uh oh... windows sockets barfed\n");
        ExitProcess(2);
    }

    // prepare the inquiry data structure
    DWORD qs_len = sizeof( WSAQUERYSET );
    WSAQUERYSET *qs = (WSAQUERYSET*) malloc( qs_len );
    ZeroMemory( qs, qs_len );
    qs->dwSize = sizeof(WSAQUERYSET);
    qs->dwNameSpace = NS_BTH;

    DWORD flags = LUP_CONTAINERS;
    flags |= LUP_FLUSHCACHE | LUP_RETURN_NAME | LUP_RETURN_ADDR;
    HANDLE h;

    // start the device inquiry
    if( SOCKET_ERROR == WSALookupServiceBegin( qs, flags, &h )) {
        ExitProcess(2);
    }
```

```
    // iterate through the inquiry results
    bool done = false;
    while(! done) {
        if(NO_ERROR == WSALookupServiceNext(h, flags, &qs_len, qs)) {
            char buf[40] = {0};
            SOCKADDR_BTH *sa =
                (SOCKADDR_BTH*)qs->lpcsaBuffer->RemoteAddr.lpSockaddr;
            BTH_ADDR result = sa->btAddr;
            DWORD bufsize = sizeof(buf);
            WSAAddressToString(qs->lpcsaBuffer->RemoteAddr.lpSockaddr,
                    sizeof(SOCKADDR_BTH), NULL, buf, &bufsize);
            printf("found: %s - %s\n",
                    buf, qs->lpszServiceInstanceName);
        } else {
            int error = WSAGetLastError();

            if( error == WSAEFAULT ) {
                free( qs );
                qs = (WSAQUERYSET*) malloc( qs_len );
            } else if( error == WSA_E_NO_MORE ) {
                printf("inquiry complete\n");
                done = true;
            } else {
                printf("uh oh.  error code %d\n", error);
                done = true;
            }
        }
    }
    WSALookupServiceEnd( h );
    free( qs );
    WSACleanup();
    return 0;
}
```

Representing Bluetooth Addresses

The Microsoft Bluetooth API represents a Bluetooth address as a 64-bit unsigned integer. There is a dedicated data type, `BTH_ADDR`, defined as

```
typedef ULONGLONG BTH_ADDR;
```

Although this representation takes up more space than the required 48 bits, using a basic data type, rather than a composite structure or array, is convenient. To convert a `BTH_ADDR` into a human-readable string, it should

be wrapped inside a SOCKADDR_BTH data structure, and then passed to WSAAddressToString. Similarly, WSAStringToAddress is used to convert string representations of Bluetooth addresses into SOCKADDR_BTH structs, which have the BTH_ADDR representation as an internal field:

```
INT WSAAddressToString( LPSOCKADDR lpsaAddress, DWORD dwAddressLength,
    LPWSAPROTOCOL_INFO lpProtocolInfo, LPTSTR lpszAddressString,
    LPDWORD lpdwAddressStringLength );

INT WSAStringToAddress( LPTSTR AddressString, INT AddressFamily,
    LPWSAPROTOCOL_INFO lpProtocolInfo, LPSOCKADDR lpAddress,
    LPINT lpAddressLength );
```

The function WSAAddressToString takes five parameters. The first is a pointer to a SOCKADDR_BTH, cast as a pointer to a SOCKADDR. Its addressFamily field should be AF_BTH, and its btAddr field should be the Bluetooth address being converted. The second parameter is always sizeof(SOCKADDR_BTH). The third parameter is always NULL, while the last two are the address and size of a string buffer, respectively. When the function returns, the string buffer is filled with a human-readable version of the specified address.

The function WSAStringToAddress also takes five parameters. The first three are the string buffer, AF_BTH, and NULL. The last two parameters are a pointer to a SOCKADDR_BTH cast as a pointer to a SOCKADDR, and a pointer to an integer whose value is sizeof(SOCKADDR_BTH). If the string is in the proper format of "XX:XX:XX:XX:XX:XX", then upon return, the address structure is populated with the specified Bluetooth address.

Starting a Device Inquiry

To start the device discovery process for detecting nearby Bluetooth devices, use the WSALookupServiceBegin function:

```
int WSALookupServiceBegin( WSAQUERYSET* queryset,
                DWORD dwControlFlags, HANDLE* lphLookup );
```

The parameter queryset points to a valid WSAQUERYSET data structure, and is initially used to specify that a Bluetooth device discovery is desired. The WSAQUERYSET structure has quite a few fields, but only two of them matter for a device inquiry. queryset->dwSize should always be set to sizeof(WSAQUERYSET), and queryset->dwNameSpace should always be set to NS_BTH. All other fields should be initialized to 0.

`lphLookup` is an output parameter that should initially point to an unused `HANDLE`. After the function returns successfully, `lphLookup` is used with other functions to refer to the ongoing device discovery process.

`dwControlFlags` specifies the exact details of the device discovery, and is a combination (bitwise OR) of the following flags:

- `LUP_CONTAINERS` is always specified for device discoveries.
- `LUP_FLUSHCACHE` flushes the cache of previously detected devices. If not specified, then devices detected in previous device discoveries may also be returned in the current one.
- `LUP_RETURN_TYPE` causes the Bluetooth device class to be retrieved. Device classes are described in the Bluetooth Assigned Numbers document.* This field can be useful for doing things like displaying a different type of icon for each type of detected device (e.g., cell phones, printers, and computers). `bthdef.h` has a number of macros useful for parsing this field.
- `LUP_RETURN_NAME` indicates that the function also attempts to determine the user-friendly name of each detected device.
- `LUP_RETURN_ADDR` retrieves the Bluetooth address of each detected device. You almost certainly want to specify this.

Parsing Device Inquiry Results

`WSALookupServiceBegin` only initiates the device discovery, and doesn't return any information about nearby devices. To find out which devices were actually detected, use `WSALookupServiceNext`. Each call to this function retrieves details for at most one device (possibly none), so it should be called repeatedly until there are no more devices left to detect:

```
int WSALookupServiceNext( HANDLE hLookup, DWORD dwControlFlags,
                          DWORD* lpdwBufferLength, WSAQUERYSET* queryset );
```

The first two parameters are related to the call to `WSALookupServiceBegin`. `hLookup` and `dwControlFlags` should be the handle created by and the flags passed to `WSALookupServiceBegin`, respectively. `lpdwBufferLength` should point to a `DWORD` containing the size, in bytes, of the `queryset`. This is also used as an output parameter in special cases, described shortly. Finally, `queryset` is an output parameter that should point to a valid `WSAQUERYSET`

* https://www.bluetooth.org/foundry/assignnumb/document/baseband

structure. It could (but doesn't have to) be the same structure that was passed to WSALookupServiceBegin. When WSALookupServiceNext completes successfully, the following fields of queryset will take meaning:

- lpszServiceInstanceName: If the LUP_RETURN_NAME flag was specified, this points to a character string containing the user-friendly name of the device.
- lpServiceClassId: If LUP_RETURN_TYPE was specified, this points to a GUID structure whose Data1 field contains the device class of the detected device.
- lpcsaBuffer: If LUP_RETURN_ADDR was specified, this points to a CSADDR_INFO structure, whose RemoteAddr field contains a SOCKET_ADDRESS structure, whose lpSockaddr field points to a SOCKADDR_BTH structure, whose btAddr field is a BTH_ADDR representing the address of the detected device. Confused? It might be easier to just look at the examples!

Occasionally, WSALookupServiceNext will fail and return an error code of WSAEFAULT. This just indicates that queryset isn't large enough to hold the information about the detected device. In this case, lpdwBufferLength will be filled in with the size needed to hold everything. Simply reallocate a new pointer with at least that amount of space and issue a new call to WSALookupServiceNext. You should expect this to happen, and have your program handle it accordingly.

Finishing a Device Discovery

When all the detected devices have been reported, WSALookupServiceNext will fail with an error code of WSA_E_NO_MORE. The program should then end the inquiry with a call to WSALookupServiceEnd and continue:

```
int WSALookupServiceEnd( HANDLE hLookup);
```

This function simply releases the resources used during the device discovery process. It should be passed the same handle used in the other two functions.

4.3 RFCOMM Sockets

Microsoft follows the same socket programming conventions used in other APIs and programming languages, including the same functions for creating, connecting, and transferring data with sockets. The socket addressing data structure and the integer constants, however, are unique. Examples 4.2 and 4.3 demonstrate how to establish RFCOMM connections and transfer data.

They also serve as examples for using the Service Discovery Protocol (SDP), which is covered in the next section.

Following standard procedure, use the `socket` function to allocate a Bluetooth socket:

```
SOCKET socket( int af, int type, int protocol );
```

Since the Microsoft Bluetooth API supports only RFCOMM sockets, the three parameters to this function will always be `AF_BTH`, `SOCK_STREAM`, and `BTHPROTO_RFCOMM`.

Addressing Structure

Next, a `SOCKADDR_BTH` is filled in to specify the details of either which device to connect to (for client applications) or which RFCOMM port to listen on (for server applications):

```
typedef struct _SOCKADDR_BTH {
    USHORT addressFamily;
    BTH_ADDR btAddr;
    GUID serviceClassId;
    ULONG port;
} SOCKADDR_BTH;
```

The first field, `addressFamily`, should always be `AF_BTH`. The second field, `btAddr`, has different meanings for listening and cilent sockets. For listening sockets, it should generally be 0 to indicate that the use of any local Bluetooth adapter is fine. If the application needs to listen on a specific Bluetooth adapter, then `btAddr` should be the address of that adapter. For client sockets, `btAddr` should be the address of the device to connect to. The third field, `serviceClassId`, is not used. Finally, `port` should be the RFCOMM port number to listen on or connect to.

The `SOCKADDR_BTH` data structure is used with many socket functions as either an input or output parameter, and will always be cast as a `SOCKADDR` structure. Since socket programming is used for many different kinds of network programming, the socket addressing structures used vary widely. The structure `SOCKADDR` can be thought of as a generic socket addressing structure that is cast into a specific structure when needed. In object-oriented terminology, `SOCKADDR` is like an abstract superclass of the different types of addressing structures.

Establishing a Connection

An application can either establish an outgoing connection using the `connect` function or accept incoming connections with the trio of functions `bind`, `listen`, and `accept`:

```
int connect( SOCKET sock, const SOCKADDR* name, int namelen );
int bind( SOCKET sock, const SOCKADDR* name, int namelen );
int listen( SOCKET sock, int backlog );
int accept( SOCKET sock, SOCKADDR* addr, int* addrlen );
```

All four of these functions behave the same as their Linux counterparts, described in Section 3.2. In each function, `sock` should be a socket created by a call to `socket`.

The function `connect` attempts to establish an outgoing connection with the specified device. The parameter `name` should be a pointer to a `SOCKADDR_BTH` cast as a `SOCKADDR*`, and should be filled in with the address and port number of the target device. The `namelen` parameter is always `sizeof(SOCKADDR_BTH)`.

The function `bind` reserves resources on a local Bluetooth adapter. The parameter `name` should be filled in with the address of the local Bluetooth adapter to use and a port number. The address can be 0 to use any available Bluetooth adapter, and the port number can be `BT_PORT_ANY` to use a dynamically assigned port.

The function `listen` switches the socket into listening mode, and can only be invoked after a call to `bind`. The parameter `backlog` is usually set to 1, but can be higher if the application expects to accept many incoming connections in a short period of time.

The function `accept` waits for an incoming connection, and can only be invoked after a call to `listen`. The output parameter `addr` will be filled in with the address and port number of the connected device when `accept` completes successfully. `addrlen` is an output parameter that will be set to `sizeof(SOCKADDR_BTH)`.

When a server application uses a dynamically assigned port number by binding to `BT_PORT_ANY`, it's sometimes useful to know which port is actually being used. To get this information, the application can invoke `getsockname` after a successful call to `bind`:

```
int getsockname( SOCKET sock, SOCKADDR* name, int* namelen );
```

sock should be the bound socket, name should point to an unused SOCKADDR_BTH, and namelen should point to an integer with value sizeof(SOCKADDR_BTH). When getsockname returns, name will be populated with the resources used by the bound socket.

Using a Connected Socket

Once a connection is established, the application can transfer data using the send and recv functions, and terminate the connection with closesocket:

```
int send( SOCKET s, const char* buf, int len, int flags );
int recv( SOCKET s, char* buf, int len, int flags );
int closesocket( SOCKET s );
```

The functions send and recv each take four parameters, the first being a connected Bluetooth socket. For send, the next two parameters should be a pointer to a buffer containing the data to send and the amount of the buffer to send, in bytes. For recv, the second two parameters should be a pointer to a buffer into which incoming data will be copied and an upper limit on the amount of data to receive. The last parameter, flags, should be set to 0 for normal operation in both send and recv.

The function send returns the number of bytes actually transmitted, which may be smaller than the amount requested. In that case, the program should just try again starting from where send left off. Similarly, recv returns the number of bytes actually received, which may be smaller than the amount requested. The special case where recv returns 0 indicates that the connection is broken and no more data can be transmitted or received.

When a socket is no longer needed, an application should call closesocket on the socket to disconnect it and free the system resources used by that connection. Note that there's a slight name difference between closesocket, used in the Microsoft Bluetooth API, and close, used in Linux.

Example 4.2 Windows RFCOMM server using SDP.

```
#include "stdafx.h"
#include <initguid.h>
#include <winsock2.h>
#include <ws2bth.h>

DEFINE_GUID(SAMPLE_UUID, 0x31b44148, 0x041f, 0x42f5, 0x8e, \
        0x73, 0x18, 0x6d, 0x5a, 0x47, 0x9f, 0xc9);
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    SOCKET server;
    SOCKADDR_BTH sa;
    int sa_len = sizeof(sa);

    // initialize windows sockets
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested = MAKEWORD( 2, 0 );
    if( WSAStartup( wVersionRequested, &wsaData ) != 0 ) {
        ExitProcess(2);
    }

    // create the server socket
    server = socket(AF_BTH, SOCK_STREAM, BTHPROTO_RFCOMM);
    if( SOCKET_ERROR == server ) {
        ExitProcess(2);
    }

    // bind the server socket to an arbitrary RFCOMM port
    sa.addressFamily = AF_BTH;
    sa.btAddr = 0;
    sa.port = BT_PORT_ANY;
    if(SOCKET_ERROR == bind(server,
                (const sockaddr*)&sa, sizeof(SOCKADDR_BTH))) {
        ExitProcess(2);
    }

    listen( server, 1 );

    // check which port we're listening on
    if(SOCKET_ERROR == getsockname(server, (SOCKADDR*)&sa, &sa_len)) {
        ExitProcess(2);
    }
    printf("listening on RFCOMM port: %d\n", sa.port);

    // advertise the service
    CSADDR_INFO sockInfo;
    sockInfo.iProtocol = BTHPROTO_RFCOMM;
    sockInfo.iSocketType = SOCK_STREAM;
    sockInfo.LocalAddr.lpSockaddr = (LPSOCKADDR) &sa;
    sockInfo.LocalAddr.iSockaddrLength = sizeof(sa);
    sockInfo.RemoteAddr.lpSockaddr = (LPSOCKADDR) &sa;
```

```
    sockInfo.RemoteAddr.iSockaddrLength = sizeof(sa);

    WSAQUERYSET svcInfo = { 0 };
    svcInfo.dwSize = sizeof(svcInfo);
    svcInfo.dwNameSpace = NS_BTH;
    svcInfo.lpszServiceInstanceName =
        "Win32 Sample Bluetooth Service";
    svcInfo.lpszComment = "Description of service...";
    svcInfo.lpServiceClassId = (LPGUID) &SAMPLE_UUID;
    svcInfo.dwNumberOfCsAddrs = 1;
    svcInfo.lpcsaBuffer = &sockInfo;

    if( SOCKET_ERROR ==
            WSASetService( &svcInfo, RNRSERVICE_REGISTER, 0 ) ) {
        ExitProcess(2);
    }

    SOCKADDR_BTH ca;
    int ca_len = sizeof(ca);
    SOCKET client;
    char buf[1024] = { 0 };
    DWORD buf_len = sizeof(buf);

    client = accept( server, (LPSOCKADDR) &ca, &ca_len );

    if( SOCKET_ERROR == client ) {
        ExitProcess(2);
    }

    WSAAddressToString((LPSOCKADDR)&ca, (DWORD) ca_len, NULL, buf,
            &buf_len);
    printf("Accepted connection from %s\n", buf);

    int received = 0;
    received = recv( client, buf, sizeof(buf), 0 );
    if( received > 0 ) {
        printf("received: %s\n", buf);
    }

    closesocket(client);
    closesocket(server);
    return 0;
}
```

Example 4.3 Windows RFCOMM client using SDP.

```
#include "stdafx.h"
#include <initguid.h>
#include <winsock2.h>
#include <ws2bth.h>

DEFINE_GUID(SAMPLE_UUID, 0x31b44148, 0x041f, 0x42f5, 0x8e, \
        0x73, 0x18, 0x6d, 0x5a, 0x47, 0x9f, 0xc9);

int SDPGetPort( const char *addr, LPGUID guid )
{
    int port = 0;
    HANDLE h;
    WSAQUERYSET *qs;
    DWORD flags = 0;
    DWORD qs_len;
    bool done;

    qs_len = sizeof(WSAQUERYSET);
    qs = (WSAQUERYSET*) malloc( qs_len );
    ZeroMemory( qs, qs_len );
    qs->dwSize = sizeof(WSAQUERYSET);
    qs->lpServiceClassId = guid;
    qs->dwNameSpace = NS_BTH;
    qs->dwNumberOfCsAddrs = 0;
    qs->lpszContext = (LPSTR) addr;
    flags = LUP_FLUSHCACHE | LUP_RETURN_ADDR;

    if( SOCKET_ERROR == WSALookupServiceBegin( qs, flags, &h )) {
        ExitProcess(2);
    }

    done = false;
    while ( ! done ) {
        if( SOCKET_ERROR ==
                WSALookupServiceNext(h, flags, &qs_len, qs) ) {
            int error = WSAGetLastError();
            if( error == WSAEFAULT ) {
                free(qs);
                qs = (WSAQUERYSET*) malloc( qs_len );
            } else if (error == WSA_E_NO_MORE ) {
                done = true;
            } else {
                ExitProcess(2);
```

```
            }
        } else {
            SOCKADDR_BTH *sa =
                (SOCKADDR_BTH*)qs->lpcsaBuffer->RemoteAddr.lpSockaddr;
            port = sa->port;
        }
    }
    free(qs);

    WSALookupServiceEnd( h );
    return port;
}

int _tmain(int argc, _TCHAR* argv[])
{
    SOCKET sock;
    SOCKADDR_BTH sa = { 0 };
    int sa_len = sizeof(sa);

    // initialize windows sockets
    WORD wVersionRequested;
    WSADATA wsaData;
    wVersionRequested = MAKEWORD( 2, 0 );
    if( WSAStartup( wVersionRequested, &wsaData ) != 0 ) {
        ExitProcess(2);
    }

    // parse the specified Bluetooth address
    if( argc < 2 ) {
        fprintf(stderr, "usage: rfcomm-client <addr>\n"
                "\n  addr must be in the form (XX:XX:XX:XX:XX:XX)");
        ExitProcess(2);
    }
    if( SOCKET_ERROR == WSAStringToAddress( argv[1], AF_BTH,
                NULL, (LPSOCKADDR) &sa, &sa_len ) ) {
        ExitProcess(2);
    }

    // query it for the right port

    // create the socket
    sock = socket(AF_BTH, SOCK_STREAM, BTHPROTO_RFCOMM);
    if( SOCKET_ERROR == sock ) {
        ExitProcess(2);
```

```
    }

    // fill in the rest of the SOCKADDR_BTH struct
    sa.port = SDPGetPort(argv[1], (LPGUID) &SAMPLE_UUID);
    if( sa.port == 0 ) {
        ExitProcess(2);
    }

    if( SOCKET_ERROR == connect( sock, (LPSOCKADDR) &sa, sa_len ) ) {
        ExitProcess(2);
    }

    send( sock, "hello!", 6, 0 );

    closesocket(sock);

    return 0;
}
```

4.4 Service Discovery Protocol

There are two ways to work with the SDP in the Microsoft Bluetooth API – the easy way and the hard way. The easy way uses the same data structures and function calls used in device discovery, but can only work with very simple service records. The hard way involves constructing and parsing binary SDP records, but it allows programs to handle arbitrarily complex services. Code Examples 4.2 and 4.3 demonstrate the easy way of advertising and searching for services, which should be sufficient for most applications.

Advertising a service, in both the easy and hard way, uses a single function call, `WSASetService`, but with different data structures. The easy way supports only a handful of attribute/value pairs, while the hard way provides for arbitrary pairs using a binary encoding. We describe both methods, but do not explain the actual construction or parsing of binary SDP records, which is already well documented in the Bluetooth Core Specification. The description of service advertisement is followed by a description of service lookup.

> *Note:* Before setting off to write your own SDP-enabled application complete with numerous GUIDs defined all over the place, take a look at `bthdef.h`. Distributed with the Windows Platform SDK, it contains many macros and definitions commonly used in SDP.

4.4.1 Advertising a Service – The Easy Way

Advertising a service, both the easy and hard way, is accomplished by the function `WSASetService`:

```
INT WSASetService( LPWSAQUERYSET lpqsRegInfo,
    WSAESETSERVICEOP essOperation, DWORD dwControlFlags );
```

The first parameter to this function, `lpqsRegInfo`, should be a pointer to a `WSAQUERYSET` (explained below). The second parameter, `essOperation`, should always be `RNRSERVICE_REGISTER`. The third parameter, `dwControlFlags`, should always be 0.

Advertised services are automatically unadvertised when an application exits. Calling `WSASetService` with the second parameter of `RNRSERVICE_DELETE` instead of `RNRSERVICE_REGISTER` deletes the advertisement immediately, although it may remain in the cache of other devices.

The easy way can be used when only the following attribute/value pairs are needed:

- Service Name
- Service Description
- Service Class ID List*
- Protocol Descriptor List

Two data structures are used to describe a Bluetooth service. The first structure, a `CSADDR_INFO` structure, contains all the information that a remote device needs to connect to the service (see Section 4.1.4 for structure components). The first two fields, `LocalAddr` and `RemoteAddr`, should set their `lpSockaddr` subfields to point to a `SOCKADDR_BTH` containing the address and port used by the socket, and have their `iSockaddrLength` subfields set to `sizeof(SOCKADDR_BTH)`. The last two fields, `iProtocol` and `iSocketType`, should always be `BTHPROTO_RFCOMM` and `SOCK_STREAM`, respectively.

For example, if `sa` is a `SOCKADDR_BTH` with the address and port of a bound, listening socket, then we might see

```
CSADDR_INFO sockInfo;
sockInfo.iProtocol                  = BTHPROTO_RFCOMM;
```

* The Service Class ID List is further limited in that only one Service Class ID can be advertised or found using this method.

```
sockInfo.iSocketType                = SOCK_STREAM;
sockInfo.LocalAddr.lpSockaddr       = (LPSOCKADDR) &sa;
sockInfo.RemoteAddr.lpSockaddr      = (LPSOCKADDR) &sa;
sockInfo.LocalAddr.iSockaddrLength  = sizeof(sa);
sockInfo.RemoteAddr.iSockaddrLength = sizeof(sa);
```

The second data structure, `WSAQUERYSET`, is the same type used earlier to describe detected Bluetooth devices, but uses more of the fields. The additional fields should be set as follows:

- `dwSize` should always be `sizeof(WSAQUERYSET)`.
- `dwNameSpace` should always be `NS_BTH`.
- `dwNumberOfCsAddrs` should always be 1.
- `lpServiceInstanceName` should be a text string giving the name of the service.
- `lpszComment` should be a text string describing the service.
- `lpServiceClassId` should point to a single GUID (Microsoft's name for Universally Unique Identifiers (UUIDs)) specifying the Service Class of the service.
- `lpcsaBuffer` should point to the `CSADDR_INFO`, as just described.

For example,

```
WSAQUERYSET serviceInfo             = { 0 };
serviceInfo.dwSize                  = sizeof(serviceInfo);
serviceInfo.dwNameSpace             = NS_BTH;
serviceInfo.dwNumberOfCsAddrs       = 1;
serviceInfo.lpszServiceInstanceName = "Sample Bluetooth Service";
serviceInfo.lpszComment             = "Description of service...";
serviceInfo.lpServiceClassId        = (LPGUID) &SAMPLE_UUID;
serviceInfo.lpcsaBuffer             = &sockInfo;
```

Once these two structures have been filled in, advertising the service can be accomplished with a call to `WSASetService`.

4.4.2 Advertising a Service – The Hard Way

The hard way to advertise a service makes use of a binary SDP record. Applications using attributes beyond the four supported by the previous method must encode them in a binary SDP record. This can be a pretty tedious process, but it's well documented in the Bluetooth Core Specification.

> *Tip:* An easy way to get a valid binary SDP record to test with is to do a service search and save one of the binary SDP records found. This is described in the next section.

Given a binary SDP record, actually advertising it is quite simple, as demonstrated in the following code snippet:

```
int advertiseRawSdpRecord( char *record, int recordlen, HANDLE *h ) {
    WSAQUERYSET qs = { 0 };
    BLOB blob = { 0 };
    BTH_SET_SERVICE *si;
    int silen;
    ULONG sdpVersion;
    int status;

    silen = sizeof(BTH_SET_SERVICE) + recordlen - 1;
    si = (BTH_SET_SERVICE*) malloc(silen);
    ZeroMemory( si, silen );

    sdpVersion = BTH_SDP_VERSION;
    si->pSdpVersion = &sdpVersion;
    si->pRecordHandle = h;
    si->ulRecordLength = recordlen;
    memcpy( si->pRecord, record, recordlen );
    blob.cbSize = silen;
    blob.pBlobData = (BYTE*)si;

    qs.dwSize = sizeof(qs);
    qs.lpBlob = &blob;
    qs.dwNameSpace = NS_BTH;

    status = WSASetService( &qs, RNRSERVICE_REGISTER, 0 );
    free( si );
    return status;
}
```

The process here is similar to the easy way of advertising a service record, but this version uses the `BTH_SET_SERVICE` data structure. The first thing to notice is the assignment to the variable `silen`, which specifies the size of the memory allocated for the data structure. It should always be set to `sizeof(BTH_SET_SERVICE)+recordlen-1` bytes, where `recordlen` is the size of the binary SDP record. The −1 is there because the size of the data structure already includes 1 byte for the record; to make space for the full record, the

program should allocate recordlen-1 additional bytes at the end of the data structure.

```
typedef struct _BTH_SET_SERVICE {
    PULONG pSdpVersion;
    HANDLE *pRecordHandle;
    ULONG fCodService;
    ULONG Reserved[5];
    UCHAR pRecord[1];
} BTH_SET_SERVICE, *PBTH_SET_SERVICE;
```

This data structure is used to store the binary service record; the record itself comes at the end of the structure; that is, the pRecord field. The header fields of BTH_SET_SERVICE are as follows: pSdpVersion should always point to an integer with value BTH_SDP_VERSION, and pRecordHandle should point to an unused HANDLE. If the service is successfully advertised, then pRecordHandle is used as an output parameter to store the handle of the advertised record. This handle can be used at some later point to modify and delete the record. Reserved should always be set to 0, and fCodService can either be left at zero or set to a Bluetooth Device Class.

Once the BTH_SET_SERVICE structure is filled in, it gets wrapped inside a BLOB; for example,

```
typedef struct _BLOB {
    ULONG cbSize;
    BYTE* pBlobData;
} BLOB;
```

The BLOB is nothing more than a wrapper structure that stores a pointer to the BTH_SET_SERVICE in its pBlobData field and the size of the BTH_SET_SERVICE structure in the cbSize field.

Finally, several fields of the WSAQUERYSET structure need to be set. lpBlob points to the BLOB, dwNameSpace is set to NS_BTH, and dwSize is set to sizeof(WSAQUERYSET). The function WSASetService uses these structures to actually advertise the service record.

To stop advertising the service record, create the same exact data structures again, but with the pRecordHandle field of the BTH_SET_SERVICE structure pointing to the HANDLE that was filled in by the original call to advertise the service. A call to the function WSASetService with RNRSERVICE_DELETE deletes the service record ensuring it is no longer advertised.

4.4.3 Searching for Services

Applications can search for specific services provided by specific Bluetooth devices, not only to see if the service exists but also to know the port associated with the service. This subsection describes how to search a specific Bluetooth device for a service. If the application doesn't know which device has its desired service, the application could conduct a device discovery to detect all nearby devices and then search each individual device.

With the Microsoft Bluetooth API, a service search is similar to a device discovery. The major differences are how the `WSAQUERYSET` data structure is used, and the flags passed to `WSALookupServiceBegin` and `WSALookupServiceNext`. To start off, allocate a `WSAQUERYSET` and set the following fields:

- `dwSize` should always be `sizeof(WSAQUERYSET)`.
- `dwNameSpace` should always be `NS_BTH`.
- `dwNumberOfCsAddrs` should always be 0.
- `lpszContext` should be a string containing the Bluetooth address of the device to search, as created by `WSAAddressToString`. It is also possible to search the local service records by specifying the address of a local Bluetooth adapter. Note that it should *not* be a `BTH_ADDR` but a string.
- `lpServiceClassId` should point to the `GUID` of the desired service. Note that it does not have to be a Service Class ID, and that any service with any UUID matching this value will be returned. For example, If the application wants a list of all services available, then this should point to `PublicBrowseGroupServiceClass_UUID`:

```
WSAQUERYSET qs       = { 0 };
qs.dwSize            = sizeof(WSAQUERYSET);
qs.dwNameSpace       = NS_BTH;
qs.dwNumberOfCsAddrs = 0;
qs.lpszContext       = "00:11:22:33:44:55";
qs.lpServiceClassId  = &PublicBrowseGroupServiceClass_UUID;
```

Next, initiate the service search by calling `WSALookupServiceBegin`.

```
int WSALookupServiceBegin( WSAQUERYSET* queryset,
                DWORD dwControlFlags, HANDLE* lphLookup );
```

The parameter `queryset` should point to the `WSAQUERYSET` just created. `dwControlFlags` should be a logical OR of `LUP_FLUSHCACHE` and `LUP_RETURN_ALL`.

`lphLookup` is an output parameter that should point to an unused `HANDLE`, which will be filled in after a successful call to the function.

To iterate through the search results, the application makes repeated calls to `WSALookupServiceNext` the same way it did for device discovery:

```
int WSALookupServiceNext( HANDLE hLookup, DWORD dwControlFlags,
                DWORD* lpdwBufferLength, WSAQUERYSET* queryset );
```

As with device discovery, `hLookup` should be the value of the handle created by `WSALookupServiceBegin`, `dwControlFlags` should be the same flags passed in to `WSALookupServiceBegin`, and `lpdwBufferLength` should point to a `DWORD` containing the size, in bytes, of `queryset`.

On each successful call, the `WSAQUERYSET` passed in will be populated as follows:

- `lpszServiceInstanceName`: the name of the service advertised;
- `lpszComment`: a description of the service;
- `lpcsaBuffer`: a pointer to a `CSADDR_INFO` structure, whose `RemoteAddr` field contains details on the protocol and port number to use for connecting to the service;
- `lpServiceClassId`: points to the first Service Class ID in the matching service record; and
- `lpBlob`: a pointer to a `BLOB` that stores the binary SDP record in its `pBlobData` field.

In some cases, `WSALookupServiceNext` will fail with an error code of `WSAEFAULT` to indicate that the `WSAQUERYSET` passed in is not large enough. As before, reallocate the data structure and try again.

As with device discovery, after the service is found or there are no more advertised services, the lookup should be ended so that resources can be released:

```
int WSALookupServiceEnd( HANDLE hLookup );
```

4.5 Summary

Hopefully, you've finished this chapter with an understanding of the concepts and considerations involved in Bluetooth programming for Microsoft Windows. It can be a bit hairy because of the driver and API issues, and

applications that need to run on older versions of Windows, such as Windows 98, NT, or 2000, will need to be written using a third-party Bluetooth stack. Apart from that, we've seen that all the concepts and techniques we've encountered still apply, and Bluetooth applications that only need the RFCOMM transport protocol and basic SDP support are easy to create with the Microsoft Bluetooth API. Although this may seem like a limited feature set, quite a few Bluetooth applications fall into this category.

This chapter has not covered all the available functions and data structures in the Microsoft Bluetooth API. There are a number of auxiliary functions that are useful for working with Bluetooth security, setting the discoverability and availability of a system, and lower level operations that are well documented in the Windows Platform SDK. Once you've mastered the basics introduced in this chapter, it's definitely a good idea to at least skim through those to know what's available.

Although we haven't mentioned it yet, Bluetooth programming for the Pocket PC/Windows Mobile environment is very similar in nature to its Windows XP counterpart. As with Windows XP, there are two major competing Bluetooth stacks available for Pocket PC – the Widcomm stack and the Microsoft stack. Unlike Windows XP, the user rarely has the ability to change the stack by installing a different set of drivers. As a result, a Pocket PC application developer has to make the choice of supporting one stack at the expense of alienating Pocket PC platforms running a different stack, or writing an application to support both stacks. Both the Widcomm and Microsoft Bluetooth APIs for Pocket PC differ slightly from their Windows XP counterparts, but the primary data structures, classes, and functions remain the same. Perhaps surprisingly, the Microsoft Bluetooth stack for Pocket PC is more comprehensive than is the Microsoft Windows XP Bluetooth stack, and provides support for L2CAP and HCI communications.

Java – JSR 82

Device discovery
 	
Name lookup
	Implement the DiscoveryListener interface
	`MyDiscoveryListener lstnr = MyDiscoveryListener();`
	`LocalDevice local = LocalDevice.getLocalDevice();`
	`DiscoveryAgent agent = local.getDiscoveryAgent();`
	`agent.startInquiry( DiscoveryAgent.GIAC, lstnr);`

SDP connect SDP search	Implement the DiscoveryListener interface `agent.searchServices(null, uuids, device, lstnr );`

Establish an outgoing connection	`StreamConnection conn = (StreamConnection)`
	`  Connector.open( "btspp://address:port" );`
	
	`L2CAPConnection conn = (L2CAPConnection)`
	`  Connector.open( "btl2cap://address:port" );`

Advertise an SDP service
 	
Establish an incoming connection
	`Connection svr = Connector.open(`
	`  "btspp://address:uuid;options");`
	`StreamConnection conn = (StreamConnection)`
	`  ((StreamConnectionNotifier)svr).acceptAndOpen();`
	
	`Connection svr = Connector.open(`
	`  "btl2cap://address:uuid;options");`
	`L2CAPConnection conn = (L2CAPConnection)`
	`  ((L2CAPConnectionNotifier)svr).acceptAndOpen();`

Transfer data	`conn.send( data );` `conn.receive( buffer );`
Disconnect	`conn.close();`

JSR-82 Quick Reference

Chapter 5

Java – JSR-82

It seems that the Java programming community has a strong desire to have a standardized API for everything under the sun, Bluetooth included. With the ratification of the Bluetooth specification, the Java community banded together and hammered out an API for Bluetooth programming that is now known as JSR-82.* Following the "write-once-run-anywhere" mantra, JSR-82 was designed so that any Java program using this API could be run on any device or environment that supports JSR-82. Unfortunately, such universality is rarely achieved, and Bluetooth is no exception.

Our goal is to present the essentials of JSR-82, using the techniques and concepts introduced in earlier chapters. We believe that with this foundation, interested readers can then tackle the full, in-depth API found in the JSR-82 specification. We do not give the complete descriptions of classes, member variables, and methods, but just the essential ones that are supported in JSR-82.

* A group whose members represented 21 companies began defining the API in December 2000 under the Java Community Process. This effort was given the designation Java Specification Request 82, or JSR-82. Drafts of the JSR-82 specification, also known as the Java APIs for Bluetooth wireless technology (JABWT), were made available for public review in the fourth quarter of 2001, and the 1.0 version of the specification was released in March 2002. In other words, JSR-82 has nothing to do with 1982. The full specification can be found at `http://www.jcp.org/aboutJava/communityprocess/final/jsr082/.`

Table 5.1 Transport protocols supported by JSR-82

Java - JSR82	RFCOMM	L2CAP	SCO	HCI

Both the RFCOMM and L2CAP transport protocols are supported in JSR-82 (see Table 5.1). JSR-82 also specifies an API for the OBEX protocol, but in our experience this is often omitted from an implementation. For example, many (if not most) Nokia cellular phones that support JSR-82 do not implement OBEX support. Support for L2CAP may also vary, especially in Windows XP implementations that use the Microsoft Bluetooth stack (see Chapter 4 for details). Since there is no central authority that grants permission for the use of the term "JSR-82," we recommend checking the supported features on the specific implementations for a target device to ensure that the necessary functionality is available.

> *Note:* The example applications in this section work best on a desktop or laptop computer. (Windows XP, OS X, and GNU/Linux should all be fine.) For brevity and clarity, we've left out all of the user interface code that would normally accompany a Java program written for cell phones or other handheld devices.

Although the essentials are the same, there are several notable differences. The next section highlights these differences. In addition, the process of device and service discovery are more tightly integrated than in the other language and system implementations covered in previous chapters (see Sections 5.2 and 5.3). Another difference is that as each device or service is detected, a callback is invoked. A significantly different way of specifying devices is also something to pay attention to while reading Sections 5.4 and 5.5 on RFCOMM and L2CAP sockets.

5.1 Notable Differences

Several functional aspects of JSR-82 stand out from the other Bluetooth programming environments we've seen so far, and are worth mentioning before we dive in:

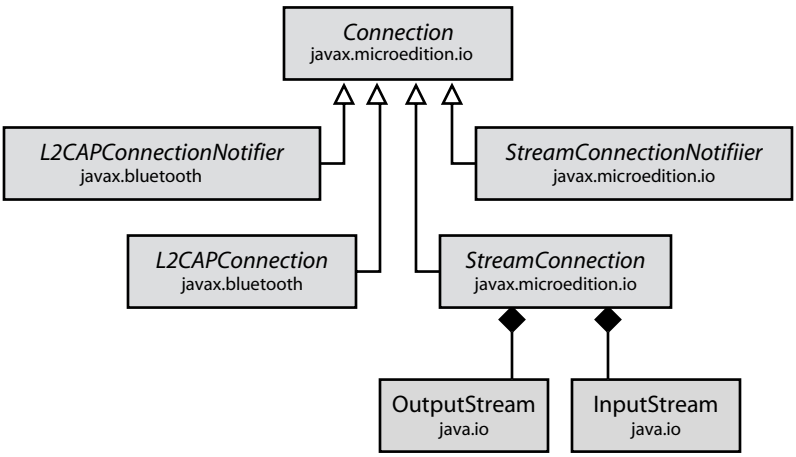


Figure 5.1 Inheritance and containment diagram for Java Bluetooth connections. Empty triangular arrows denote inheritance, and solid diamonds indicate containment.

Sockets and connections: In an attempt to be somewhat more intuitive, what we've been referring to all along as sockets are referred to in Java as *Connections*. For example, a connected RFCOMM socket is represented by the `StreamConnection` interface, and a connected L2CAP socket is represented by the `L2CAPConnection` interface, both of which implement the `Connection` interface. To be consistent, we'll still call them sockets, but note that other JSR-82-related documents will probably call them connections.

Additionally, listening sockets are no longer the same data type as connected sockets. For example, a listening RFCOMM socket corresponds to a `StreamConnectionNotifier`, and a listening L2CAP socket corresponds to a `L2CAPConnectionNotifier`. The inheritance hierarchy of `connection` classes is illustrated in Figure 5.1, and will be discussed in more detail later on.

Service Discovery Protocol (SDP): It is not possible in JSR-82 to create a listening socket without also advertising an SDP service. At least one Service Class ID must be specified when creating a listening socket. The service record can be modified after the corresponding listening socket has been created, but it cannot be unadvertised before the socket is closed.

Dynamically assigned port numbers only: Listening sockets in JSR-82 are always dynamically assigned port numbers, and it is not possible to use

programmer-assigned port numbers. This may come across as an inconvenience for those who just want to put up a quick Bluetooth service, although it does encourage "proper" usage of port numbers. As a consequence, Bluetooth client applications connecting to services implemented in JSR-82 must use SDP to find the port number used by the service.

Tighter integration of Bluetooth security: Bluetooth security is tightly integrated into JSR-82, and options for Bluetooth authentication and encryption are often built into the various connection methods. We'll see this in the examples, but the basic idea is that JSR-82 forces the programmer to consider Bluetooth security in places where other Bluetooth programming environments do not.

5.2 Device Detection

Device discovery and service search have much in common, so we present only a single integrated sample code in Example 5.1. This section is concerned with device discovery of publicly detectable devices, while service search and advertisement are explained in the next section.

The process of detecting nearby devices can be outlined as follows:

(i) Get a reference to the singleton instance of `LocalDevice`.
(ii) Get a reference to the `DiscoveryAgent` of the `LocalDevice`.
(iii) Use the `DiscoveryAgent` to start a device discovery via its `startInquiry` method.

Example 5.1 JSR-82 device inquiry.

```
import javax.microedition.io.*;
import javax.bluetooth.*;

public class ServiceBrowser implements DiscoveryListener {
    static int SERVICE_NAME_ATTRID = 0x0100;
    RemoteDevice discovered[] = new RemoteDevice[255];
    int num_discovered;

    public void deviceDiscovered( RemoteDevice rd, DeviceClass cod ) {
        String addr = rd.getBluetoothAddress();
        String name = "";
        try {
            name = rd.getFriendlyName(true);
```

```
        } catch( java.io.IOException e ) {}

        this.discovered[this.num_discovered] = rd;
        this.num_discovered++;
        System.out.println("discovered " + addr + " - " + name);
    }

    public void inquiryCompleted( int status ) {
        System.out.println("device inquiry complete");
        synchronized(this) {
            try { this.notifyAll(); } catch(Exception e) {}
        }
    }

    public void servicesDiscovered( int transID, ServiceRecord[] rec ){
        for( int i=0; i<rec.length; i++ ) {
            DataElement d =
                rec[i].getAttributeValue( SERVICE_NAME_ATTRID );
            if ( d != null ) System.out.println((String)d.getValue());
            else System.out.println("unnamed service");

            System.out.println(rec[i].getConnectionURL(
                    ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false));
        }
    }

    public void serviceSearchCompleted( int transID, int respCode ) {
        System.out.println("service search complete");
        synchronized(this) {
            try { this.notifyAll(); } catch(Exception e) {}
        }
    }

    public static void main(String argv[]) {
        try {
            ServiceBrowser sb = new ServiceBrowser();
            LocalDevice ld = LocalDevice.getLocalDevice();
            DiscoveryAgent da = ld.getDiscoveryAgent();

            // device inquiry
            sb.num_discovered = 0;
            da.startInquiry( DiscoveryAgent.GIAC, sb );
            synchronized(sb) {
                try { sb.wait(); } catch( Exception e ) {}
```

```
            }

            // service search
            UUID uuids[] = new UUID[1];
            uuids[0] = new UUID( 0x1002 );
            int attridset[] = new int[1];
            attridset[0] = SERVICE_NAME_ATTRID;

            for( int i=0; i<sb.num_discovered; i++ ) {
                RemoteDevice rd = sb.discovered[i];
                da.searchServices(attridset, uuids, rd, sb);
                synchronized(sb) {
                    try { sb.wait(); } catch( Exception e ) {}
                }
            }
        } catch( BluetoothStateException e ) {
            System.out.print(e.toString());
        }
    };
};
```

(iv) Each time a device is detected, the `deviceDiscovered` method of a `DiscoveryListener` is invoked, with the details of the detected device as parameters.

(v) When the device discovery has finished, the `inquiryComplete` method of a `DiscoveryListener` is invoked.

A local Bluetooth adapter is represented by the `LocalDevice` class. JSR-82 assumes that there is only one local Bluetooth adapter, and the singleton instance of this adapter is retrieved by invoking the static method `getLocalDevice()`.

```
LocalDevice ld = LocalDevice.getLocalDevice();
```

The `LocalDevice` class can be used to obtain and set information about the local device, and update advertised service records. Additionally, the `getDiscoveryAgent` method retrieves a reference to a `DiscoveryAgent`, whose sole purpose is to conduct device inquiries and SDP searches. To start a device inquiry, use the `startInquiry` method of the `DiscoveryAgent`:

```
DiscoveryAgent da = ld.getDiscoveryAgent();
da.startInquiry( DiscoveryAgent.GIAC, sb );
```

The first parameter of `startInquiry` is almost always set to the constant `DiscoveryAgent.GIAC` to instruct the agent to search for all generally discoverable devices. In rare cases, different values can be used to find devices responding to specialized access codes. The second parameter, `sb`, is a class implementing the `DiscoveryListener` interface.

The function `startInquiry` initiates the device inquiry and returns immediately. Detected devices and the rest of the device inquiry are handled by `sb`, which contains several callback methods. Two of these, `deviceDiscovered` and `inquiryComplete`, are used in the device inquiry process. The other two, `servicesDiscovered` and `serviceSearchCompleted`, are used for SDP and are described in the next section.

```
public void deviceDiscovered( RemoteDevice rd, DeviceClass cod ) {
    String addr = rd.getBluetoothAddress();
    ...
}
public void inquiryCompleted( int status ) {
    System.out.println("device inquiry complete");
    ...
}
```

Each time a device is discovered, `deviceDiscovered` is invoked. A detected Bluetooth device is represented by the `RemoteDevice` class. Two of the most useful methods of this class are `getBluetoothAddress` and `getFriendlyName`, which return the Bluetooth address and display name of the device as strings, respectively. Keep in mind that `getFriendlyName` involves actually connecting to the remote device again and requesting its display name, so it can take some time or even fail if the device becomes unresponsive or unavailable.

The `cod` parameter of `deviceDiscovered` represents the device class, and is a convenience class with accessor methods for pulling out that information.

> *Note:* JSR-82 does not have a data structure for explicitly representing Bluetooth addresses. Instead, when they are used, they appear as 12-character strings consisting of the 12 hexadecimal digits of a Bluetooth address. The only difference between this representation and the Python representation is the lack of colons delimiting each byte of the address.

When the inquiry finishes, cancels, or otherwise fails after being started, `inquiryCompleted` is invoked with a single parameter describing the condition that caused the inquiry to complete. `status` will take on value `INQUIRY_COMPLETED`, `INQUIRY_TERMINATED`, or `INQUIRY_ERROR` if the inquiry completed successfully, was canceled by the user, or some other error occurred, respectively.

5.3 Service Discovery

Conducting a service search uses the same classes as a device discovery. In JSR-82, no convenience functions are provided for searching all nearby devices, so the program must specifically perform a search on each one. Our example program, Example 5.1, maintains a list of all detected devices. When the device inquiry is complete, it then iterates through each detected device and searches it for all publicly advertised services (using the reserved Public Browse Group Universally Unique Identifier (UUID) of `0x1002`).

To begin a service search on a single device, use the `searchServices` method of the `DiscoveryAgent`:

```
UUID uuids[] = new UUID[1];
uuids[0] = new UUID( 0x1002 );
int attridset[] = new int[1];
attridset[0] = SERVICE_NAME_ATTRID;

for( int i=0; i<sb.num_discovered; i++ ) {
    RemoteDevice rd = sb.discovered[i];
    da.searchServices(attridset, uuids, rd, sb);
    ...
}
```

The function `searchServices` takes four parameters that specify the attributes to return, the UUIDs to search for, the remote device to search, and the `DiscoveryListener` that will handle the results of the search. Each parameter requires some explanation.

Recall from Section 1.2.4 that a service record is a list of attribute-value pairs. In the other Bluetooth APIs we've seen, service searches return all the attribute-value pairs for the service records found in a search. JSR-82 takes a different approach and retrieves only a few of the attribute-value pairs by

default; any additional desired attributes must be manually specified. The attributes that are always retrieved are

- Service Record Handle;
- Service Class ID List;
- Service Record State;
- Service ID; and
- Protocol Descriptor list.

If the first parameter to `searchServices` is `null`, then only the defaults are returned. Otherwise, each additional attribute specified in the parameter is retrieved for all matching service records. In this example, the Service Name, with attribute ID `0x0100`, is requested.

The second parameter is an array of UUIDs to search on. `searchServices` will only find services matching every UUID in the array. In our example, the Public Browse Group reserved UUID `0x1002` is used, which means to search for all publicly browseable services.

The last two parameters are fairly straightforward, and are simply a `RemoteDevice` found during the device inquiry and the same `DiscoveryListener` used to conduct the device inquiry. This time, the callback methods `servicesDiscovered` and `serviceSearchCompleted` are used to handle the results of a service search:

```
public void servicesDiscovered(int transID, ServiceRecord[] rec) {
    for( int i=0; i<rec.length; i++ ) {
        DataElement d =
            rec[i].getAttributeValue( SERVICE_NAME_ATTRID );
        if ( d != null ) System.out.println((String)d.getValue());
        else System.out.println("unnamed service");

        System.out.println(rec[i].getConnectionURL(
                ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false));
    }
}

public void serviceSearchCompleted( int transID, int respCode ) {
    System.out.println("service search complete");
    synchronized(this) {
        try { this.notifyAll(); } catch(Exception e) {}
    }
}
```

The function `servicesDiscovered` is invoked each time one or more services are found. The first parameter, `transID`, is an identifier also returned by `searchServices`. On some devices, it's possible to conduct multiple service search in parallel, but generally only one at a time is expected. The second parameter, `rec`, is an array of `ServiceRecord` objects, each of which represents a matching service record.

The `ServiceRecord` class is a data structure, with several useful access methods. The most useful is the `getConnectionURL` method. It returns a string containing the Bluetooth address and port number of the device and service that can later be used to establish a connection. The first parameter can be either `NOAUTHENTICATE_NOENCRYPT`, `AUTHENTICATE_NOENCRYPT`, or `AUTHENTICATE_ENCRYPT` to specify desired levels of Bluetooth security. The second parameter should usually be `false`, and `true` only when the local device absolutely must be the master of the piconet when connecting the specified service. Another `ServiceRecord` method, `getAttributeValue`, is useful when displaying search results in a user interface, as it retrieves the data elements stored within the service record. Our example attempts to retrieve and display the Service Name attribute.

5.4 RFCOMM Sockets

The code in Examples 5.2 and 5.3 demonstrate how to accept and initiate RFCOMM connections, respectively. The server application advertises the Serial Port Service Class ID. For brevity, the client application is hard-coded to connect to "01:23:45:67:89:AB" on port 1, although a more robust application would use the device detection and service search process described in the previous section.

Incoming Connections

Establishing an incoming connection requires first defining a URL string, which specifies some basics of both how to listen for incoming connection requests and the service record to advertise. There is a header followed by a list of attribute-value pairs. For example,

```
String url = "btspp://localhost:" +
    new UUID( 0x1101 ).toString() +
    ";name=SampleServer";
```

The URL string for RFCOMM always starts with "btspp://" and is followed by the address of the local Bluetooth adapter to use, or "localhost" if it doesn't

Example 5.2 JSR-82 RFCOMM Server

```
import java.io.*;
import javax.microedition.io.*;
import javax.bluetooth.*;

public class RFCOMMServer {

    public static void main( String args[] ) {
        try {
            String url = "btspp://localhost:" +
                new UUID( 0x1101 ).toString() +
                ";name=SampleServer";
            StreamConnectionNotifier service =
                (StreamConnectionNotifier) Connector.open( url );

            StreamConnection con =
                (StreamConnection) service.acceptAndOpen();
            OutputStream os = con.openOutputStream();
            InputStream is = con.openInputStream();

            String greeting = "JSR-82 RFCOMM server says hello";
            os.write( greeting.getBytes() );

            byte buffer[] = new byte[80];
            int bytes_read = is.read( buffer );
            String received = new String(buffer, 0, bytes_read);
            System.out.println("received: " + received);

            con.close();
        } catch ( IOException e ) {
            System.err.print(e.toString());
        }
    }
}
```

Example 5.3 JSR-82 RFCOMM Client.

```
import java.io.*;
import javax.microedition.io.*;
import javax.bluetooth.*;

public class RFCOMMClient {

    public static void main( String args[] ) {
        try {
            String url = "btspp://0123456789AB:3";

            StreamConnection con =
                (StreamConnection) Connector.open(url);

            OutputStream os = con.openOutputStream();
            InputStream is = con.openInputStream();

            String greeting = "JSR-82 RFCOMM client says hello";
            os.write( greeting.getBytes() );

            byte buffer[] = new byte[80];
            int bytes_read = is.read( buffer );
            String received = new String(buffer, 0, bytes_read);
            System.out.println("received: " + received);

            con.close();
        } catch ( IOException e ) {
            System.err.print(e.toString());
        }
    }
}
```

matter. This is followed by a colon and the a Service Class ID to advertise in the service record (more UUIDs can be added later). Next, up to five optional parameters can be specified. They appear as “attribute=value" pairs, each preceded by a semicolon. Valid options are as follows:

name: The Service Name to advertise in the service record. If not specified, then the JSR-82 implementation may use an arbitrary name.
authenticate: If set to `true`, then connecting devices must complete the Bluetooth authentication process before the connection is allowed to be established. Default is `false`.
encrypt: If set to `true`, then all established connections are encrypted according to the Bluetooth encryption process. Default is `false`.
authorize: If set to `true`, then either a database or the user is prompted to accept incoming connections before they are actually established. How or whether this actually happens is implementation specific. Default is `false`.
master: If set to `true`, then the local adapter always insists on being the master of piconets associated with incoming connections. Default is `false`.

`StreamConnectionNotifier` is the equivalent of a listening socket, and is required to accept incoming connections. A URL string, such as the one defined above, is passed to the static method `Connector.open` to instantiate the listening socket:

```
StreamConnectionNotifier service =
    (StreamConnectionNotifier) Connector.open( url );
```

A program waits for and accepts an incoming connection using the `acceptAndOpen` method of the instantiated `StreamConnectionNotifier` object:

```
StreamConnection con = (StreamConnection) service.acceptAndOpen();
```

The method `acceptAndOpen` blocks and waits for an incoming connection. When a new connection has been accepted and established, it returns a `StreamConnection` object that can be used to exchange data with the remote Bluetooth device.

Outgoing Connections

As with accepting an incoming connection, the first step in initiating an outgoing connection is to construct a URL string, either manually or extracted from a `ServiceRecord`.

The header part is similar to incoming connections, starting with "btspp://" followed by the Bluetooth address to connect to, a colon (":"), and the

RFCOMM port number the server is listening on. Three optional parameters can be specified, each preceded by a semicolon. They are as follows:

master: If set to `true`, then the client device will insist on becoming the master of any piconet formed between it and the server device. This option defaults to `false` and is rarely needed.
encrypt: This is identical to the `encrypt` option for listening sockets.
authenticate: This is identical to the `authenticate` option for listening sockets.

A typical connection string might look as follows:

```
String url = "btspp://0123456789AB:1;master=false";
```

Rather than constructing the URL string directly, if this step in an application happens subsequent to a device inquiry and service search, then the application can use the `ServiceRecord` passed into the `servicesDiscovered` method. For example, the URL can be gotten as follows:

```
public void servicesDiscovered( int transID,
        ServiceRecord[] records ) {
    ServiceRecord match = pickDesiredServiceRecord( records );
    String url = match.getConnectionURL(
        ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false ) );
}
```

No matter how the string is constructed, it is used to establish the connection when it is passed to the function `Connector.open`, which will return a `StreamConnection` on success or throw an `IOException` on failure:

```
StreamConnection con = (StreamConnection) Connector.open( url );
```

Using a Connected Socket

A `StreamConnection` represents a connected RFCOMM socket, but can't be directly used to send or receive data. Instead, instances of two other classes must be used, which can be obtained by invoking the `openOutputStream` and `openInputStream` methods of a `StreamConnection`:

```
StreamConnection con = (StreamConnection) Connector.open( url );
OutputStream os = con.openOutputStream();
InputStream is = con.openInputStream();
```

As their names suggest, `OutputStream` and `InputStream` are used to send and receive data across a connection, respectively.

To send data, pass a `byte` array to the `write` method of `OutputStream`. The function `write` attempts to transmit as much of the `byte` array as possible and returns the number of bytes it actually sent. Typically, an application will check to see if the entire array was sent, and retransmit the unsent parts:

```
OutputStream os = con.openOutputStream();
String greeting = "JSR-82 RFCOMM client says hello";
int bytes_sent = os.write( greeting.getBytes() );
```

To receive data, pass an unused `byte` array to the `read` method of `InputStream`. `read` returns the number of bytes successfully received and written into the `byte` array. If the buffer was not filled up by the read, then the remaining bytes in the array are left unchanged:

```
InputStream is = con.openInputStream();
byte read_buffer[] = new byte[80];
int bytes_received = is.read( read_buffer );
```

Once a `StreamConnection` is no longer needed, terminate the connection with the `close` method:

```
con.close();
```

If an error occurs while sending or receiving data, then both `read` and `write` throw an `IOException`. This could happen if the two devices stray out of range, if one side closes the connection early, and so on.

5.5 L2CAP Sockets

Working with L2CAP sockets is similar to working with RFCOMM sockets. There are differences in the URL string and port specifiers, as well as the methods to transmit and receive data. The code in Examples 5.4 and 5.5 demonstrate how to accept and initiate L2CAP connections, respectively. The server advertises the (poorly chosen) service class ID `0xABCD`, and the client is hardcoded to connect to the device "01:23:45:67:89:AB" on port `0x1101`.

Incoming Connections

The process for establishing an incoming L2CAP connection is almost the same as for accepting an RFCOMM connection. First, a few basics must be specified in a URL string:

```
String url = "btl2cap://localhost:" +
    "0000000000000000000000000000ABCD" +
    ";name=JSR82_ExampleService";
```

For the L2CAP protocol, the URL strings always start with "`btl2cap://`" followed by either "`localhost`" or the address of the local Bluetooth adapter to use, and then a separating colon (":"). Next, a UUID must be specified. As with the RFCOMM server URL string, this UUID is advertised in the Service Class ID list of the SDP service record that will accompany this service. Finally, the same optional parameters available to incoming RFCOMM URL strings (Section 5.4) can be used here.

> *Note:* Notice that the UUID is written out in full form as
>
> `"0000000000000000000000000000ABCD"`
>
> whereas in the RFCOMM example, we used the `UUID` class instead. We could also do that here:
>
> `new UUID( 0xABCD ).toString()`
>
> Both ways are fine, and it's up to you to decide which you prefer.

Example 5.4 JSR-82 L2CAP Server.

```
import java.io.*;
import javax.microedition.io.*;
import javax.bluetooth.*;

public class L2CAPServer {
    public static void main( String args[] ) {
        try {
            String url = "btl2cap://localhost:" +
                "0000000000000000000000000000ABCD" +
                ";name=JSR82_ExampleService";

            L2CAPConnectionNotifier service =
```

```
                (L2CAPConnectionNotifier) Connector.open( url );
            L2CAPConnection con =
                (L2CAPConnection) service.acceptAndOpen();

            String greeting = "JSR-82 L2CAP server says hello";
            con.send( greeting.getBytes() );

            byte buffer[] = new byte[80];
            int bytes_read = con.receive( buffer );
            String received = new String(buffer, 0, bytes_read);
            System.out.println("received: [" + received + "]");

            con.close();
        } catch ( IOException e ) {
            System.err.print(e.toString());
        }
    }
}
```

Once a URL string has been specified, the application can create an `L2CAPConnectionNotifier`, which is functionally identical to a bound, listening socket. The `L2CAPConnectionNotifier` can then be used to wait for and accept incoming connections with its `acceptAndOpen` method:

```
L2CAPConnectionNotifier service =
    (L2CAPConnectionNotifier) Connector.open( url );
L2CAPConnection con = (L2CAPConnection) service.acceptAndOpen();
```

Example 5.5 JSR-82 L2CAP Client.

```
import java.io.*;
import javax.microedition.io.*;
import javax.bluetooth.*;

public class L2CAPClient {
    public static void main( String args[] ) {
        try {
            String url = "btl2cap://000eed3d1829:1101";

            L2CAPConnection con =
                (L2CAPConnection) Connector.open(url);

            String greeting = "JSR-82 L2CAP client says hello";
            con.send( greeting.getBytes() );
```

```
            byte buffer[] = new byte[80];
            int bytes_read = con.receive( buffer );
            String received = new String(buffer, 0, bytes_read);
            System.out.println("received: " + received);

            con.close();
        } catch ( IOException e ) {
            System.err.print(e.toString());
        }
    }
}
```

When an incoming connection has been established, `acceptAndOpen` returns an `L2CAPConnection` that can be used to communicate with the connected device.

Outgoing Connections

Initiating an outgoing L2CAP connection is also similar to its RFCOMM counterpart, with the main difference being the class names. The first step is, once again, to obtain a URL string that specifies the details of the target device. If this step follows a service search, then the `getConnectionURL` method of a `ServiceRecord` can be used. Otherwise, the string can be manually specified.

```
String url = "btl2cap://0123456789AB:1101";
```

The URL string for an outgoing L2CAP connection always starts with "`btl2cap://`" followed by the address of the target device, a colon separator, and the port number. Note that *L2CAP port numbers must be given in hexadecimal notation*. Finally, the same optional parameters available for outgoing RFCOMM URL strings can also be used for outgoing L2CAP URL strings (Section 5.4).

To establish the connection, simply pass the URL string to the static method `Connector.open()`:

```
L2CAPConnection con = (L2CAPConnection) Connector.open( url );
```

On success, this returns a `L2CAPConnection` that can be used for further communication. On failure, it throws an `IOException`.

Using a connected socket

Unlike the `StreamConnection` used for RFCOMM, the `L2CAPConnection` can be directly used for sending and receiving data:

```
L2CAPConnection con;

// accept or initiate the connection

String greeting = "hello!";
con.send( greeting.getBytes() );

byte buffer[] = new byte[80];
int bytes_read = con.receive( buffer );
```

The two methods `send` and `receive` behave as expected. `send` takes an array of `bytes` as input, transmits it to the connected device, and returns the number of bytes successfully sent. `receive` takes an unused `byte` buffer as input, fills it with received data, and returns the number of bytes written to the buffer.

Once the connection is no longer needed, the `close` method is used to terminate the connection and free system resources:

```
con.close();
```

5.6 Summary

As with the rest of the Java programming language, Bluetooth programming in Java has the distinct advantage of a single API for a wide variety of platforms. Unfortunately (as with the rest of Java), the functionality available may vary significantly across implementations and platforms. This problem aside, however, JSR-82 is a mature and robust API that can be often used to create a large range of Bluetooth applications where other solutions are not available. Applications that require only the RFCOMM or L2CAP transport protocols, and do not require access to the HCI layer, are well suited for JSR-82.

Other platforms and environments

Device discovery Name lookup SDP connect SDP search	`bt_discover()`
Establish an outgoing connection	`s = socket(AF_BT, SOCK_STREAM)` `s.connect( ( address, port ) )`

Establish an incoming connection	`s = socket(AF_BT, SOCK_STREAM)`
	`s.bind( ( address, port ) )`
	`set_security(s, ...)`
	`s.listen( backlog )`
	`s.accept()`

Advertise an SDP service	`bt_advertise_service( s, name, ... )`
Transfer data	`s.send( data )` `s.recv( numbytes )`
Disconnect	`s.close()`

Chapter 6

Other Platforms and Environments

This chapter addresses some other platforms and environments. Although the Bluetooth specification is large and wide ranging, specifying everything from frequency hopping algorithms used during communications to the higher level communications protocols on Bluetooth printers, it does not specify a common software API. What this means, unfortunately, is that Bluetooth programming is different from platform to platform, and each platform has its own API.

The chapter consists of three sections. The first addresses mobile devices (cell phones) under the Symbian operating system. We have already addressed Bluetooth under Windows and Windows Mobile; here, we show how to write Bluetooth applications using Python under Symbian. It is different than Python using PyBlueZ as describe in Chapter 2. The second section addresses Macintosh under OS X, It has very good Bluetooth programming support with extensive documentation and so we only present the essentials. Finally, the last section of this chapter shows how to map the RFCOMM Bluetooth protocol to the long-established serial port connections on many platforms. For many simple applications, the serial port interface is the most platform-independent mechanism.

Regardless of platform, the concepts introduced in this book will always apply. The processes for device discovery, service search, and connection establishment always follow the same principles. Once you figure out how to express these essentials in the API, the rest is detail. So, even if you are not

Table 6.1 Transport protocols supported by Series 60 Python.

Series 60 Python	RFCOMM	L2CAP	SCO	HCI
Series 60 C++	RFCOMM	L2CAP	SCO	HCI

interested in the specifics of this chapter, it can be used as a guide to getting started with any new platform.

6.1 Symbian OS/Nokia Series 60

Bluetooth is currently available on nearly all middle- and high-end mobile phones. Symbian is a popular operating system for many smart phones, especially those made by Nokia. Moreover, Symbian Series 60 devices are amiable to third-party applications. Programmers wishing to create Bluetooth applications for Series 60 devices have three options for programming environments: C++, Java, or Python. The Symbian C++ programming environment affords the most control and flexibility over the device, but is also the most time consuming and difficult method to learn, partly because it is an extension of standard C++. The second option for many Series 60 devices is the Java Bluetooth API, which was described in Chapter 5. Java has some restrictions on mobile devices that may preclude its use for your intended application. The third option and the one covered in this section is Symbian Series 60 Python. It became available in late 2004 and has since evolved into a powerful yet accessible way to create Bluetooth-enabled programs for the Series 60 platform. Note, however, that Python under Symbian 60 has its own runtime library and for Bluetooth, it is different from PyBlueZ.

This section introduces Bluetooth programming with the Series 60 Python API, which leverages the simplicity of Python to enable rapid development of simple Bluetooth applications on the Series 60 platform. While the Series 60 Python Bluetooth API differs from the PyBlueZ API introduced in Chapter 2, the concepts and techniques are still the same.

Table 6.1 illustrates the transport protocols supported by the Series 60 Python and C++ APIs. The Python API also provides limited support for the Object Push Profile, which is not described here. The API for the (HCI) protocol in C++ is limited to a relatively small subset of the HCI specification, and thus HCI should be considered partially supported.

The examples shown in this section were tested on Series 60 Python 1.2. Newer symbian versions may require capabilities.

6.1.1 Device Detection and Service Discovery

The process of choosing a device, protocol, and port number to connect to in Series 60 Python is quite different from what we've seen before. In other environments, the steps of detecting nearby devices, looking up their user-friendly names, and determining which services are available are all separate. In Series 60 Python, all three of these happen at once. The code in Example 6.1 demonstrates how this is done.

Note that even though this example is only a handful of lines, it is a complete, working program. Most of the code involves setting up the user interface (menus, title bar, event handlers, etc.), and only the last portion deals with Bluetooth. The `socket` module in Series 60 Python has been extended to encompass Bluetooth functionality, and all functions and constants used in this section and the next are found in `socket`. The function that we'll focus on for now is `bt_discover`:

```
target_device, services = socket.bt_discover( address = None )
```

Although it may look straightforward, `bt_discover` is actually a bit of a mess. The Series 60 Python developers had a very clear idea in their minds of how they wanted Python programmers to use the phone's Bluetooth capabilities, so this function is highly specialized.

If the `address` parameter contains a Bluetooth address, the function then looks up available services advertised by that device and returns a single tuple with two entries.

Example 6.1 Series 60 device inquiry.

```
import e32
from appuifw import app
import socket

quit_signal = e32.Ao_lock()
old_ui_handlers = app.menu, app.exit_key_handler, app.body, app.title

def on_exit(self):
    global quit_signal, old_handlers
    app.menu, app.exit_key_handler, app.body, app.title = old_handlers
    quit_signal.signal()
```

```
app.exit_key_handler = on_exit
app.body = appuifw.Text()
app.menu = []
app.title = u"Bluetooth Inquiry"

address, services = socket.bt_discover()

appuifw.app.body.add( u"Services found on %s\n" % address )
for name, port in services.items():
    appuifw.app.body.add( u"%s : %d\n" % (name, port) )

quit_signal.wait()
```

If the optional parameter `address` is not specified, then `bt_discover` conducts a device inquiry to detect nearby devices and also tries to determine their user-friendly names. As devices are detected, they are displayed on the screen in a pop-up list, and the user is prompted to choose one device. All this happens automatically. The function returns the address-service tuple associated with the device selected by the user.

The first item in the returned tuple is simply the Bluetooth address of the device selected by the user. Similar to PyBlueZ, Series 60 Python represents Bluetooth addresses as strings of the form "XX:XX:XX:XX:XX", where each "X" is a hexadecimal digit. The second item is a dictionary of some (but not all) services advertised by the chosen device. Keys in the dictionary correspond to service names, and the values correspond to the RFCOMM port number the service is listening on. Only services advertising both the Serial Port Service Class ID (16-bit reserved Universally Unique Identifier (UUID) of `0x1101`) and the Serial Port Profile show up in this dictionary, which is important to consider when creating applications on other platforms.

Keep in mind that while using only the Series 60 Python libraries, it is not possible for a program to noninteractively detect nearby devices or to detect services that don't advertise the Serial Port Profile. Programs requiring either of these will need to make use of native extension modules, which are C++ libraries built to appear as Python modules. The Series 60 C++ development kit is distributed on the Forum Nokia Web site.*

* `http://forum.nokia.com`

6.1.2 Bluetooth Sockets

Using sockets for Bluetooth communication in Series 60 Python is a straightforward extension of the concepts we've encountered in previous chapters. The code in Examples 6.2 and 6.3 show how to use RFCOMM server sockets and client sockets, respectively.

Looking at these examples, you'll find nothing surprising at all. In both cases, the `socket` function is used to allocate a socket:

```
sock = socket.socket( family, type, protocol = BTPROTO_RFCOMM )
```

To create a Bluetooth RFCOMM socket, `family` should always be `AF_BT`, `type` should always be `SOCK_STREAM`, and `protocol` can either be left out or set to `BTPROTO_RFCOMM`. All these are defined in the `socket` class.

The Bluetooth socket object returned by `socket` behaves like a standard Python socket, and its methods share the same functionality as its PyBluez counterpart (described in Chapter 2). Since they have already been introduced several times over, we won't bother describing the following methods: `bind`, `listen`, `accept`, `connect`, `recv`, `send`, and `close`. Instead, we'll focus on the subtle differences between Series 60 Python Bluetooth sockets and PyBluez sockets.

Example 6.2 Series 60 Python RFCOMM Server.

```
import e32
import appuifw
from socket import *

quit_signal = e32.Ao_lock()
app = appuifw.app
old_handlers = app.menu, app.exit_key_handler, app.body, app.title

def on_exit():
    app.menu, app.exit_key_handler, app.body, app.title = old_handlers
    quit_signal.signal()

app.exit_key_handler = on_exit
tb = appuifw.Text()
app.body = None
app.menu = []
app.title = u"RFCOMM server"

sock = socket(AF_BT, SOCK_STREAM)
port = bt_rfcomm_get_available_server_channel(sock)
```

```
sock.bind(("", port))
set_security(sock, AUTHOR)
sock.listen(1)
bt_advertise_service(u"SampleServer", sock, True, RFCOMM)

tb.add( u"listening on port ", port )
client, addr = sock.accept()
tb.add( u"accepted connection from ", addr )
appuifw.note(u"accepted connection", "info")
sock.close()

client.send("Series 60 Python server says Hello!!")
data = client.recv(80)
tb.add( u"received: ", data )
client.close()

quit_signal.wait()
```

Example 6.3 Series 60 Python RFCOMM Client.

```
import e32
import appuifw
from socket import *

quit_signal = e32.Ao_lock()
app = appuifw.app
old_handlers = app.menu, app.exit_key_handler, app.body, app.title

def on_exit():
    app.menu, app.exit_key_handler, app.body, app.title = old_handlers
    quit_signal.signal()

app.exit_key_handler = on_exit
tb = appuifw.Text()
app.body = tb
app.menu = []
app.title = u"RFCOMM server"

address, services = bt_discover()

sock = socket(AF_BT, SOCK_STREAM)
port = services["SampleServer"]
sock.connect( (address, port) )
```

```
tb.add( u"connected to %s\n" % address )

sock.send("Series 60 Python client says Hello!!")
data = sock.recv(80)
tb.add( u"received: %s" % data )
sock.close()

quit_signal.wait()
```

Dynamically Assigned Ports

Server sockets should use `bt_rfcomm_get_available_server_channel` to dynamically choose an RFCOMM port number:

```
port = socket.bt_rfcomm_get_available_server_channel( sock )
```

This function does exactly what the name says, and should be passed a newly created RFCOMM socket. The returned port can then be passed to `bind`.

Socket Security

Unlike other Bluetooth programming environments we've seen, Series 60 Python Bluetooth server sockets are required to set a security mode. A server socket is not permitted to randomly accept connections from other devices. Incoming connections must come from previously recognized and paired devices, or the user must manually accept connections. To configure the security mode for a socket, use the `set_security` function:

```
set_security( sock, mode )
```

Here, `sock` should be a bound socket and `mode` should be a combination (bitwise OR) of `AUTH`, `ENCRYPT`, or `AUTHOR`. The flags have the following meaning:

AUTH: Connecting devices must be paired with the phone before the incoming connection is accepted. If a connecting device is not already paired during a connection attempt, then the pairing procedure is automatically initiated. If this procedure does not successfully complete, then the connection attempt is rejected.

ENCRYPT: All incoming connections accepted with this socket are encrypted. This implies `AUTH`.

AUTHOR: The user is prompted to manually accept all incoming connections.

At least one of these flags must be set. As with the device inquiry procedure, it is possible to get around this restriction by using native extension modules.

Advertising Services with SDP

Once a socket is bound and listening, `bt_advertise_service` is used to advertise the service with the phone's Service Discovery Protocol (SDP) server:

```
bt_advertise_service( sock, service_name, advertise, class )
```

There are four parameters: `sock` should be the server socket; `service_name` should be a unicode string giving the name of the service; `advertise` should be `True` to advertise the service, and `False` to stop advertising; and `class` can be either `RFCOMM` or `OBEX`. If RFCOMM is specified, then the Serial Port Profile is added to the service record, and for `OBEX`, the Object Push Profile is added instead.

> *Note:* Most text strings in Series 60 devices are internally handled as unicode strings. To convert a standard string to unicode, either use the `unicode` function or add a "u" in front of a string literal (see the example).

6.2 OS X

With its tightly controlled development environment, Macintosh's OS X has provided a unified and consistent API for Bluetooth development since OS X 10.1.3. Along with the rest of its developer tools, the Bluetooth development kit for OS X is freely distributed at `http://connect.apple.com`. This kit provides extensive documentation and examples on using both the C and Objective C languages for Bluetooth development. A full introduction to Bluetooth programming in OS X is not given here. Instead, we'll provide a short overview of how Bluetooth programming in OS X compares with the other environments described, and give two short examples that show what Bluetooth programming in OS X looks like.

Table 6.2 illustrates the transport protocols supported by the OS X Bluetooth API.

Although it does not provide direct access to an HCI connection with the local Bluetooth adapter, the OS X Bluetooth API provides a large number

Table 6.2 Transport protocols supported by the OS X Bluetooth API.

OS X	RFCOMM	L2CAP	SCO	HCI

of functions corresponding to specific HCI commands. Extensive support for the OBEX protocol is also provided.

The examples in this section were tested on OS X v10.4 (Tiger).

6.2.1 Event-Driven API

A major difference between Bluetooth programming in OS X and other programming environments is that the OS X Bluetooth API is highly asynchronous and event driven. Historically, graphical user applications almost always use an event-driven framework with a single event loop that waits for user input and dispatches event handlers accordingly. In OS X, the Bluetooth API was designed around the event loop, with the API designers clearly anticipating application developers to structure their programs in a very specific manner.

As a result, it is actually not possible for an application to easily perform certain Bluetooth operations synchronously. For example, there is no function that starts a device inquiry and returns when the inquiry is complete. There is also no function that listens for an incoming connection and returns when a connection has been established. A few operations have both synchronous and asynchronous counterparts, but for the most part, it is expected that the OS X Bluetooth developer use the OS X event loop.

6.2.2 Device Discovery

Example 6.4 demonstrates how to asynchronously detect nearby Bluetooth devices, using the Objective C Bluetooth bindings. While this method is not recommended by the Apple developer User Interface guidelines, it is consistent with the other examples on device discovery found throughout this book.

In an effort to keep the code short, this example does not use the traditional `NSApplicationMain` method of starting an application. Instead, `CFRunLoop` and `NSAutoreleasePool` structures are used to provide the application with an event loop and memory management, respectively.

Example 6.4 OS X Device Discovery.

```
#import <Foundation/NSObject.h>
#import <IOBluetooth/objc/IOBluetoothDevice.h>
#import <IOBluetooth/objc/IOBluetoothDeviceInquiry.h>
#import <stdio.h>

@interface Discoverer: NSObject {}
-(void) deviceInquiryComplete: (IOBluetoothDeviceInquiry*) sender
                        error: (IOReturn) error
                        aborted: (BOOL) aborted;
-(void) deviceInquiryDeviceFound: (IOBluetoothDeviceInquiry*) sender
                        device: (IOBluetoothDevice*) device;
@end
@implementation Discoverer
-(void) deviceInquiryComplete: (IOBluetoothDeviceInquiry*) sender
                        error: (IOReturn) error
                        aborted: (BOOL) aborted
{
    printf("inquiry complete\n");
    CFRunLoopStop( CFRunLoopGetCurrent() );
}
-(void) deviceInquiryDeviceFound: (IOBluetoothDeviceInquiry*) sender
                        device: (IOBluetoothDevice*) device
{
    printf("discovered %s\n", [[device getAddressString] cString]);
}
@end

int main( int argc, const char *argv[] )
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    Discoverer *d = [[Discoverer alloc] init];

    IOBluetoothDeviceInquiry *bdi =
        [[IOBluetoothDeviceInquiry alloc] init];
    [bdi setDelegate: d];

    [bdi start];

    CFRunLoopRun();

    [bdi release];
    [d release];
    [pool release];
    return 0;
}
```

The primary class used to conduct a device discovery in this example is `IOBluetoothDeviceInquiry`, which allows an application to noninteractively detect nearby Bluetooth devices. For a "typical" Bluetooth application that prompts the user to select a device from a list of recently detected Bluetooth devices, Apple recommends using the `IOBluetoothUI` framework instead of `IOBluetoothDeviceInquiry`. However, `IOBluetoothDeviceInquiry` behaves more like the methods we've seen in other chapters, so we're using it in this example:

```
IOBluetoothDeviceInquiry *bdi =
    [[IOBluetoothDeviceInquiry alloc] init];
[bdi setDelegate: d];
[bdi start];
```

There are three steps involved in starting a device inquiry. The first is to simply create an instance of `IOBluetoothDeviceInquiry`. The second is to assign a delegate object. Various methods of the delegate are invoked when a new device is detected, the inquiry finishes, and so on. In this example, a custom class named `Discoverer` serves as the delegate, and implements two methods of the `IOBluetoothDeviceInquiryDelegate` category:

```
@interface Discoverer: NSObject {}
-(void) deviceInquiryComplete: (IOBluetoothDeviceInquiry*) sender
                        error: (IOReturn) error
                        aborted: (BOOL) aborted;
-(void) deviceInquiryDeviceFound: (IOBluetoothDeviceInquiry*) sender
                        device: (IOBluetoothDevice*) device;
@end
```

When a new device is detected, the `deviceInquiryDeviceFound` method is invoked, with the original `IOBluetoothDeviceInquiry` as the first parameter and the newly detected device as the second. When the device inquiry completes, the `deviceInquiryComplete` method is invoked. In this example, the application exits when the device inquiry completes. Additional methods can be implemented to detect more events, and are described in the Apple Bluetooth API documentation.

Finally, to actually kick off the device inquiry, invoke the `start` method of the `IOBluetoothDeviceInquiry` instance. This method starts the inquiry and returns immediately.

6.2.3 Outgoing RFCOMM Connection

Example 6.5 demonstrates how to asynchronously establish an outgoing RFCOMM connection and use it to transmit and receive data. As with the previous example, `CFRunLoop` and `NSAutoreleasePool` structures are used to provide the application with an event loop and memory management, respectively.

Example 6.5 OS X RFCOMM Client.

```
#import <Foundation/NSObject.h>
#import <IOBluetooth/objc/IOBluetoothDevice.h>
#import <IOBluetooth/objc/IOBluetoothDeviceInquiry.h>
#import <IOBluetooth/objc/IOBluetoothRFCOMMChannel.h>
#import <stdio.h>

@interface ConnectionHandler : NSObject {}
- (void)rfcommChannelOpenComplete:(IOBluetoothRFCOMMChannel*)channel
                           status:(IOReturn)status;
- (void)rfcommChannelData:(IOBluetoothRFCOMMChannel*)channel
                     data:(void *)dataPointer
                   length:(size_t)dataLength;
@end
@implementation ConnectionHandler
- (void)rfcommChannelOpenComplete:(IOBluetoothRFCOMMChannel*)channel
                           status:(IOReturn)status
{

    if( kIOReturnSuccess == status ) {
        printf("connection established\n");
        [channel writeSync: "Hello!" length: 6];
    } else {
        printf("Connection error!\n");
        CFRunLoopStop( CFRunLoopGetCurrent() );
    }
}
- (void)rfcommChannelData:(IOBluetoothRFCOMMChannel*)channel
                     data:(void *)dataPointer
                   length:(size_t)dataLength
{
    printf("received: %s\n", dataPointer);
    [channel closeChannel];
    CFRunLoopStop( CFRunLoopGetCurrent() );
}
```

```
@end

int main( int argc, const char *argv[] )
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    NSString *addr_str = @"00:10:60:A8:95:49";
    BluetoothDeviceAddress addr;
    IOBluetoothNSStringToDeviceAddress( addr_str, &addr );
    IOBluetoothDevice *remote_device =
        [IOBluetoothDevice withAddress:&addr];
    IOBluetoothRFCOMMChannel *chan;
    ConnectionHandler *handler = [[ConnectionHandler alloc] init];

    [remote_device openRFCOMMChannelAsync:&chan withChannelID:1
                                 delegate: handler];

    CFRunLoopRun();

    [handler release];
    [pool release];
    return 0;
}
```

As its name suggests, the `BluetoothDeviceAddress` data structure is used by OS X to represent a Bluetooth address. To convert from a human-readable string to a `BluetoothDeviceAddress`, use the `IOBluetooth NSStringToDeviceAddress` function:

```
BluetoothDeviceAddress addr;
NSString *addr_str = @"00:10:60:A8:95:49";
IOBluetoothNSStringToDeviceAddress( addr_str, &addr );
```

In the socket programming model used in earlier chapters, an unconnected socket is first created and then connected to a remote device. OS X abandons this model and instead uses the `IOBluetoothDevice` class as the starting point for establishing an outgoing connection. Instances of this class can be obtained through the result of a device inquiry, as seen in code Example 6.4. They can also be manually constructed from an address using the `withAddress` static method, as seen in this example:

```
IOBluetoothDevice *remote_device =
    [IOBluetoothDevice withAddress:&addr];
```

The `openRFCOMMChannelAsync` method of a `IOBluetoothDevice` can be used to establish an outgoing connection:

```
[remote_device openRFCOMMChannelAsync:&chan withChannelID:1
                                delegate: handler];
```

This method returns immediately, and relies on the event loop to inform the application when the connection attempt succeeds or fails. The first argument is the address of an unused `IOBluetoothRFCOMMChannel*`. If the connection is successful, then this argument is set to point to a newly allocated `IOBluetoothRFCOMMChannel` that represents the new connection. The second argument specifies the RFCOMM port number of the target device. The third argument specifies a delegate, various methods of which are invoked when events related to the RFCOMM connection occur.

In this example, a custom class `ConnectionHandler` is defined and used as the delegate for the RFCOMM connection. It implements two methods of the `IOBluetoothRFCOMMChannelDelegate` interface:

```
- (void)rfcommChannelOpenComplete:(IOBluetoothRFCOMMChannel*)channel
                           status:(IOReturn)status;
- (void)rfcommChannelData:(IOBluetoothRFCOMMChannel*)channel
                     data:(void *)dataPointer
                   length:(size_t)dataLength;
```

When the connection attempt is complete (both success and failure), the first method, `rfcommChannelOpenComplete`, is invoked. The first parameter to this method simply points to the relevant connection, and the second is a status code indicating whether or not the connection attempt succeeded. On success, the status code is set to `kIOReturnSuccess`.

In this example, as soon as a connection is successfully established, the application uses `writeSync` to transmit a short message:

```
[channel writeSync: "Hello!" length: 6];
```

The function `writeSync` is similar to the `write` functions we've seen in other chapters, and simply takes as parameters a pointer to the data to send and the number of bytes to transmit. It blocks until the data is successfully transmitted, or until an error occurs. A nonblocking method `writeAsync` is also available.

When incoming data has been received, the `rfcommChannelData` method of the delegate is invoked. It is not possible to synchronously wait for incoming data; there is no equivalent of the `recv` function. The `rfcommChannelData`

method is passed three parameters, which are a pointer to the channel in question, a pointer to the received data, and the length of the received data.

To close a connection, invoke the `closeChannel` method:

```
[channel closeChannel];
```

This method takes no parameters, and simply disconnects the RFCOMM connection.

6.3 Serial Ports – Bluetooth Programming Made Easy

Under certain circumstances, Bluetooth programming can be *really easy*. The idea here is to treat an RFCOMM connection as a serial port connection, and write programs that use serial ports instead of explicitly creating Bluetooth sockets, searching for nearby devices and services, and so on. By doing so, any program written to communicate via serial port can use an RFCOMM connection, without any changes at all. In effect, it can be so easy that no programming needs to be done at all.

So what's the catch? There are two big ones, which might actually not be that bad, depending on what you're trying to do. The first is that by treating a Bluetooth connection like a serial port, you lose all control of the lower level Bluetooth semantics. You don't get to control packet flush timeouts, packet transmission sizes, connection timeouts, or anything like that. For many user-level applications, however, that doesn't matter too much.

The second catch is that in typical serial port programming, the program assumes that the user has physically connected the serial cable from one device to the other. Even though there's no physical cable in Bluetooth, the same idea still applies. The user must create a serial port "connection" between two Bluetooth devices before the program can do anything. In applications where the communicating devices don't change over time, this isn't a big deal. In other cases, the added hassle may make it faster and simpler to use the techniques introduced in earlier chapters.

6.3.1 Serial Ports Overview

Table 6.3 shows how the basic tasks in Bluetooth programming roughly translate to Bluetooth serial ports. The relationship is not as well defined as it is in actual Bluetooth programming environments, but the principles still apply.

Table 6.3 Serial port quick reference.

Device discovery Name lookup Advertise an SDP service SDP connect SDP search	Operating system and Bluetooth stack specific. Typically performed with user-level programs and utilities.
Establish an outgoing connection	Bind a local serial port (e.g., com3 or/dev/rfcomm0) to a target device and port number Open local serial port
Establish an incoming connection	Bind a local serial port to a local port number Open local serial port
Transfer data	Write to serial port Read from serial port
Disconnect	Close serial port

The serial port configuration process is different for each operating system and Bluetooth stack, but they all share the same general steps. First, a serial port is designated as either an incoming port or an outgoing port. Incoming ports are configured with an RFCOMM port number, so that whenever an incoming Bluetooth connection is established on that RFCOMM port, all data received and transmitted on that connection is directed through the serial port instead. An incoming port can't be used to establish outgoing connections, but it can be used to accept connections from arbitrary Bluetooth devices.

An outgoing port, on the other hand, must be configured with the Bluetooth address of another device, along with an RFCOMM port number. When an application first accesses the outgoing port, the operating system creates a connection to the specified Bluetooth device on the specified RFCOMM port. Then, data can be transmitted and received by writing to and reading from the serial port.

Applications that use serial ports as Bluetooth connection wrappers are compatible with those that don't, as long as all connections used are RFCOMM connections. For example, it's possible to create a Bluetooth serial port that always connects to a Bluetooth GPS receiver. The receiver probably doesn't even have serial ports in its operating system, but can still accept connections made in this manner. That way, applications that were written to use serial port GPS receivers can use the Bluetooth-enabled receiver without any changes at all.

The next several sections provide a brief description of how to use Bluetooth serial ports with some of the more popular Bluetooth stacks.

6.3.2 Serial Ports – GNU/Linux

The primary tool for creating serial port wrappers around Bluetooth connections in Linux is the `rfcomm` command.* Serial ports created in this way will typically be named `/dev/rfcommX`, where `X` ranges from 0 to 255. This naming convention may occasionally differ across GNU/Linux distributions.

Incoming Serial Port

To set up an incoming serial port, choose an unused device file (`thru/dev/rfcomm0/dev/rfcomm255`) and an unused RFCOMM port number, and use them with the `rfcomm listen` command. For example, to connect `/dev/rfcomm0` to RFCOMM port 20, we'd use it like this:

```
# rfcomm listen /dev/rfcomm0 20
```

Once an incoming connection has been established, `rfcomm` displays a short message, and the serial port can be used for communication. If a device with address "AA:BB:CC:DD:EE:FF" is connected to our device, we might see

```
# rfcomm listen /dev/rfcomm0 20
Waiting for connection on channel 1
Connection from AA:BB:CC:DD:EE:FF to /dev/rfcomm0
Press CTRL–C for hangup
```

Now, any application can access the special device file `/dev/rfcomm0` and use it like any other serial port. The command `rfcomm listen` returns when the connection has been terminated (either by pressing `CTRL-C` or by the remote device closing the connection). Note that the serial port `/dev/rfcomm0` is only valid after a connection has been accepted and before the connection terminates, so don't try to use it outside of this time window. This restriction may change in future versions of RFCOMM.

Outgoing Serial Port

To set up an outgoing serial port, use the `rfcomm connect` command, specifying the device file to use, along with the Bluetooth address and port of the target device. When connecting `/dev/rfcomm0` to RFCOMM port 20 on the device "11:22:33:44:55:66" we might see

* The procedures described in this section were tested on `rfcomm` 2.24.

```
# rfcomm connect /dev/rfcomm0 11:22:33:44:55:66 20
Connected /dev/rfcomm0 to 11:22:33:44:55:66 on channel 20
Press CTRL–C for hangup
```

As with `rfcomm listen`, any application can now access the serial port.

A second way of using `rfcomm` to set up an outgoing serial port is with `rfcomm bind`. This has the same syntax as `rfcomm connect`, but returns immediately and doesn't establish the Bluetooth connection until some program actually tries to access the serial port. For example,

```
# rfcomm bind /dev/rfcomm0 11:22:33:44:55:66 20
```

The advantage of using `rfcomm bind` over `rfcomm connect` is that the connection is made only when the serial port is actively used by an application. Thus, the binding can be made well in advance without wasting resources. Additionally, the binding is persistent, so that opening and closing the serial port multiple times has the effect of establishing and closing a Bluetooth connection multiple times.

Used it this way, it is not necessary to invoke `rfcomm` every time a connection is established, but the binding only lasts until a reboot. To make it permanent, one can add a stanza to the `/etc/bluetooth/rfcomm.conf` configuration file so that this binding is made on system start-up. Add the following lines:

```
rfcomm0 {
    bind yes;
    device 11:22:33:44:55:66;
    channel   20;
    comment "GPS receiver";
}
```

Unfortunately, there's currently no way to add incoming serial ports to this configuration file. To set up an incoming serial port every time the system boots, a small script can be added to the start-up scripts for the particular distribution of Linux.

6.3.3 Serial Ports – Windows XP (Microsoft Stack)

> *Note:* This section describes the use of Bluetooth serial ports for the Microsoft Bluetooth stack. The Widcomm stack uses a different process, and is described in the next section.

Bluetooth serial ports in the Microsoft Bluetooth stack can be configured via the "Bluetooth Devices" widget in the Windows Control Panel.* Double-clicking this icon opens up a window with four tabs: Devices, Options, COM Ports, and Hardware. Bluetooth serial ports are managed from within the "COM Ports" tab, so start by navigating there.

Incoming Serial Port

To set up an incoming serial port, click the "Add" button within the COM Ports tab. A new dialog box should pop up titled "Add COM Port."

At the top of the dialog box, there are two radio buttons, labeled "Incoming" and "Outgoing." Select the "Incoming" radio button and click "OK." The "Add COM Port" dialog box will close, leaving the original "Bluetooth Devices" dialog box.

After a few seconds, information about the newly created incoming serial port should show up in a list inside the "COM Ports" tab. Applications can now access the Bluetooth serial port like any other serial port. This serial port is always valid and will reappear on rebooting the machine.

Prior to an application accessing a Bluetooth serial port, there is no Bluetooth activity related to that port. As soon as an application opens the port, a listening RFCOMM socket is allocated and an SDP record corresponding to that port is advertised. The serial port is writable when an incoming connection is established, which is how the application is able to detect a connection. When the application closes the serial port, all Bluetooth activity related to that port ceases – open sockets are closed, and SDP records are removed.

Outgoing Serial Port

To set up an outgoing serial port, navigate to the "COM Ports" tab of the "Bluetooth Devices" dialog box, and click the "Add" button. A new dialog box should pop up titled "Add COM Port."

This time select the "Outgoing" radio button and click "Browse."

Windows will search for nearby Bluetooth devices and display the results. Choose the device you want to set up the serial port with and select "OK." Windows will search this device for services advertising the Serial Port Profile and present a list of these services. Note that it will let you choose *only* from

* The procedures described in this section were tested on Windows XP SP 2.

Serial Port Profile services, and no others. It is not possible to choose an arbitrary port number.

Select the service you want to bind to the serial port, and then click "OK."

After a few seconds, information about the newly created outgoing serial port should show up in the "COM Ports" tab. This serial port can now be used for Bluetooth communication with the target device.

Whenever an application accesses that serial port, Windows will create an RFCOMM client socket and attempt to connect to the target device. If the connection cannot be established, then the application will not able to open the serial port.

6.3.4 Serial Ports – Windows XP (Widcomm Stack)

The process for creating a Bluetooth serial port with the Widcomm stack is similar to Microsoft Bluetooth stack counterpart. The system tray in the bottom right corner of the screen should show a Bluetooth icon that corresponds to the Widcomm stack.

Right click this icon and select "Advanced Configuration." (This can also be accessed through the Control Panel.) A dialog box titled "Bluetooth Configuration" should appear.

Incoming Serial Port

To set up an incoming serial port, activate the "Local Services" tab, and then click on the "Add Serial Service" button. A dialog box titled "Bluetooth Properties" should appear.

Optionally enter in a name to describe the serial port. This name is used in the SDP record advertising the serial port. If desired, select the "Start-up Automatically" checkbox to have the serial port activated when Windows starts. The "Secure Connection" checkbox can also be selected to force Bluetooth authentication and encryption with devices connected on that serial port. When finished, click "OK." The serial port is now ready to use.

One significant difference between incoming serial ports in the Widcomm stack and incoming serial ports in the Microsoft stack is that SDP records for incoming serial ports are always advertised in the Widcomm stack, whereas SDP records for serial ports in the Microsoft stack are advertised only when an application has opened the port.

Outgoing Serial Port

There are two steps to setting up an outgoing serial port. The first is to create the serial port, and the second is to bind it to a remote service. First, open up the same "Bluetooth Configuration" dialog box as before, but this time activate the "Client Applications" tab.

Click the "Add COM Port" button. A dialog box titled "Bluetooth Properties" should appear.

Optionally enter in a name to describe the serial port. This is mostly for your benefit. If desired, select the "Secure Connection" checkbox to force Bluetooth authentication and encryption. Click "OK" to save the serial port.

The next step is to bind the newly created serial port to a remote service. Close the open dialog boxes. Locate the system tray icon originally used to open the "Bluetooth Configuration" dialog box. Right click it and select "My Bluetooth Places." (This may also be accessible from the desktop, My Computer, or the Control Panel.) A window titled "My Bluetooth Places" should appear.

The left-hand column should show a subwindow titled "Bluetooth Tasks." From this subwindow, select "View devices in range."

If the target device is not immediately listed, then select "Search for devices in range" from the "Bluetooth Tasks" subwindow (or press F5). When the target device appears, double-click it to display the SDP records it has advertised.

Locate the desired service. Right click the service, and select "Connect to Bluetooth Serial Port." If necessary, you will be prompted to enter a PIN to pair the two devices. When finished, a small dialog window will pop up, providing information about which COM port the service was bound to. (This is automatically chosen.) The serial port is now ready to use.

The serial port binding is persistent, and needs to be completed once only. In the future, whenever an application opens the serial port, the Widcomm stack will attempt to transparently establish the Bluetooth connection.

6.3.5 Serial Ports – OS X

Bluetooth serial ports on OS X* behave much like they do in GNU/Linux, which should not be surprising, given that they are both UNIX-based

* The procedures described in this section were tested on OS X 10.4.7 (Tiger).

operating systems. Configuring Bluetooth serial ports in OS X can be done by opening the "System Preferences" window and then selecting "Bluetooth."

Incoming Serial Port

To configure an incoming serial port, select the "Sharing" tab near the top right of the "Bluetooth" window. Then, click the button labeled "Add Serial Port Service" in the bottom left corner of the window.

A few options will show up on the right-hand side that allow the new serial port to be configured. The name should describe or identify the intended usage of the serial port, and is used to actually access the serial port later on. If the connecting device is a modem, then leave the serial port type as "Modem," otherwise set it to "RS-232." If Bluetooth level security is desired, then ensure that the "Require pairing for security" option is checked.

The incoming Bluetooth serial port can then be accessed as `/dev/tty.Name`, where `Name` is the name given to the serial port during the initial set up. The serial port service is always advertised via SDP, but a remote device can establish a connection only when an application on the local computer has actively opened the proper device file.

Outgoing Serial Port

To configure an outgoing serial port, the OS X computer must first be paired with the target device. To do this, return to the "Bluetooth" window (accessed from "System Preferences") and select the "Devices" tab near the top middle of the window. Click the button labeled "Set Up New Device" in the bottom right corner of the window. Click "Continue." You will be presented with a list of device types to choose from. Serial port is not a listed option – just click "Any device" followed by "Continue" to initiate a device discovery procedure.

Ensure that the target device is discoverable, and wait for it to appear in the next window. Once the target device is listed, click "Continue." OS X will conduct an SDP search to find available services, and then prompt you to click "Continue" once again. Depending on the type of device (the advertised Class of Device), OS X will prompt you to either type in a PIN on the target device or provide OS X with a PIN. Once the pairing is complete, you may see a window with the text, "There were no supported services found on your device." This is fine, as OS X does not look for serial ports at this time. Click "Continue," and then "Quit" to complete the procedure.

You should now be back at the "Devices" tab in the "Bluetooth" window, and the target device should be listed near the top left, under the heading "Bluetooth Devices." Click the target device, and click "Edit Serial Ports."* A smaller window should pop up, with a few options. The *Port name* specifies the file name for the serial port; it will be available as `/dev/tty.portname`. If the target device advertises more than one serial port service, then choose the desired service from the "Device service" drop-down list. Next, if the target device is a modem, change "Port type" to "Modem," otherwise leave it at RS-232.

When satisfied with the settings, click "Apply." The serial port is now ready for use. When an application first opens the serial port, OS X will attempt to establish the connection. If the connection cannot be established (e.g., the target device is out of range or simply not available) then opening the serial port will fail.

6.4 Summary

With this section, we've introduced the Series 60 Python and OS X Bluetooth APIs. Both have their own particular quirks, but the main concepts and techniques introduced in earlier chapters still apply. It is unfortunate that no standardized cross-platform Bluetooth API exists, but it turns out that once the basics have been mastered, adapting to new platforms and stacks is a simple and straightforward process.

We've also seen that Bluetooth programming with serial ports is a powerful tool that is highly useful in situations where the user has the expertise and time to manually establish the Bluetooth connections, and when the communicating devices do not change frequently. The exact process for setting up a Bluetooth serial port differs across platforms, but they all have a great deal in common. However, it is also important to realize the limitations of Bluetooth serial ports and to understand the trade-offs involved when using them.

We certainly have not introduced a number of other Bluetooth stacks and platforms, but our hope is that we have provided a solid foundation upon which the reader can build.

* The target device *must* advertise its service with SDP in order for OS X to use it properly. If this is not the case, then OS X is unable to automatically configure the Bluetooth serial port.

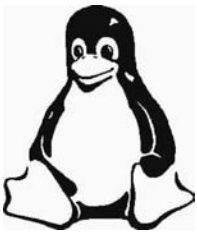
Bluetooth Tools in Linux

`hciconfig`	Configure the basic properties of local adapters
`hcitool`	Detect nearby devices; display information on and adjust low-level connections
`sdptool`	Search for and browse SDP services Basic configuration of locally advertised services
`hcidump`	Low-level debugging of connection setup and data traffic
`l2ping`	Test L2CAP connection functionality
`uuidgen`	Generates a unique UUID for use with SDP

Bluetooth Linux Tools Quick Reference

Chapter 7

Bluetooth Tools GNU/Linux



Before writing code or while debugging code, it is helpful for the programmer to be able to discover devices and services, connect to them, ping devices, and monitor communications through known working commands. Without such aids, it is difficult to know if it is one's code that is buggy or one's understanding as to how things should work. For example, a failure to find a particular service on a particular device might be due to the fact that the device advertises the service in a nonstandard way. This chapter considers the tools available in Linux; other operating systems provide some of this functionality, although often through a GUI that provides neither complete control nor a direct mapping to the operations that your program might perform.

There are three major parts of the Bluetooth subsystem in Linux – the kernel-level routines, the `libbluetooth` development library, and the user-level tools and commands. The kernel is responsible for managing the Bluetooth hardware resources that are attached to a machine, wrestling with the idiosynchrosies of each manufacturer; Bluetooth adapter, and presenting a unified interface to the rest of the system that allows any Bluetooth application to work with any Bluetooth hardware.

The `libbluetooth` development library provides a set of convenient data structures and functions that can be used by Bluetooth programmers. It abstracts some of the most commonly performed operations (such as detecting

nearby Bluetooth devices) and provides simple functions that can be invoked to perform common tasks. This was described in Chapter 3.

The user-level tools are a set of programs that perform very specific Bluetooth tasks and can be invoked from the command line. After mastering the preceding chapters, you could have written the programs to perform these tasks. It is comforting to know that someone else has already done so. The tools, however, play a function beyond pedagogy. The interaction with Bluetooth devices controlled by a program ends when the program ends. All open sockets are closed when a program terminates. Linux maintains a set of daemons that carry out the tasks specified by these tools; they are constantly running programs that use the Bluetooth development library to manage the system's Bluetooth resources in the ways configured by the user through the tools.

The BlueZ developers strive to make these tools and daemons as straightforward to use as possible, while also providing enough flexibility to meet every user's needs. As a software developer in Linux, you'll be interacting with these user-level tools, and we predict that you will make frequent use of them.

There are six command-line tools provided with BlueZ that are indispensable when configuring Bluetooth on a machine and debugging applications. Three of the command names begin with "hci" and one with "sdp." Recall that "hci" stands for `Host-Controller Interface` and "sdp" for `Service Discovery Protocol`. The tools accept a wide range of parameters and commands. Some of the commands merely return information, while others take action or change configurations. We give some short descriptions here on how they're useful, and show some examples on how to use them. For full information on their use, consult the `man` pages that are distributed with the tools, or invoke each tool with the `-h` flag. This chapter covers only the essentials.

7.1 `hciconfig`

The command `hciconfig` is used to configure the basic properties of Bluetooth adapters. As the name suggests, it provides a user-level interface to the (HCI) protocol. When invoked without any arguments, it will display the status of the adapters attached to the local machine. In all other cases, the usage follows the form:

```
# hciconfig <device> <command> <arguments...>
```

where `<device>` is usually `hci0`. (`hci1` specificies the second Bluetooth adapter if you have two, `hci2` is the third, etc.) Most of the commands require superuser privileges. Some of the most useful ways to use this tool are presented below. The command that the user types into the Linux shell is prefaced by the hash (#) prompt. This is usually followed by the output of the system.

Display the Status of Recognized Bluetooth Adapters

The most basic usage, without parameters or commands, simply supplies information:

```
# hciconfig
hci0:   Type: USB
        BD Address: 00:0F:3D:05:75:26 ACL MTU: 192:8 SCO MTU: 64:8
        UP RUNNING PSCAN ISCAN
        RX bytes:505075 acl:31 sco:0 events:5991 errors:0
        TX bytes:25758 acl:24 sco:0 commands:1998 errors:0
```

Each Bluetooth adapter recognized by BlueZ is displayed here. In this specific case, there is only one adapter, `hci0`, and it has Bluetooth Address `00:0F:3D:05:75:26`. The "UP RUNNING" part on the second line indicates that the adapter is enabled. "PSCAN" and "ISCAN" refer to Inquiry Scan and Page Scan, which are described a few paragraphs down. The rest of the output is mostly statistics and a few device properties.

Enable/Disable an Adapter

The `up` and `down` commands to this tool can be used to enable and disable a Bluetooth adapter:

```
# hciconfig hci0 down
# hciconfig
hci0:   Type: USB
        BD Address: 00:0F:3D:05:75:26 ACL MTU: 192:8 SCO MTU: 64:8
        DOWN
        RX bytes:505335 acl:31 sco:0 events:5993 errors:0
        TX bytes:25764 acl:24 sco:0 commands:2000 errors:0
# hciconfig hci0 up
# hciconfig
hci0:   Type: USB
```

```
        BD Address: 00:0F:3D:05:75:26 ACL MTU: 192:8 SCO MTU: 64:8
        UP RUNNING PSCAN ISCAN
        RX bytes:505075 acl:31 sco:0 events:5991 errors:0
        TX bytes:25758 acl:24 sco:0 commands:1998 errors:0
```

Display and Change the User-Friendly Name of an Adapter

The `name` command is fairly straightforward, and can be used to display and change the user-friendly name of the Bluetooth adapter:

```
# hciconfig hci0 name
hci0:   Type: USB
        BD Address: 00:0F:3D:05:75:26 ACL MTU: 192:8 SCO MTU: 64:8
        Name: 'Trogdor'
# hciconfig hci0 name 'StrongBad'
# hciconfig hci0 name
hci0:   Type: USB
        BD Address: 00:0F:3D:05:75:26 ACL MTU: 192:8 SCO MTU: 64:8
        Name: 'StrongBad'
```

Configure Inquiry Scan and Page Scan

The inquiry scan and page scan settings for a Bluetooth adapter determine whether it is detectable by nearby Bluetooth devices and whether it will accept incoming connection requests, respectively. The following table summarizes which commands to use when configuring these two options:

Inquiry Scan	Page Scan	Command
On	On	`piscan`
Off	On	`pscan`
On	Off	`iscan`
Off	Off	`noscan`

For example, the following invocation disables both Inquiry Scan and Page Scan for the first Bluetooth adapter:

```
# hciconfig hci0 noscan
```

hciconfig Summary

There are many more ways to use `hciconfig`, all of which are described in the help text (`hciconfig -h`) and the manpages (`man hciconfig`). It is the tool to use for any nonconnection-related settings for a Bluetooth adapter.

Finally, note that changes made by `hciconfig` are only temporary, and the effects are erased after a reboot or when the device is disabled and enabled again. The recommended way to ensure that a configuration option is always set is to modify `/etc/bluetooth/hcid.conf`. This file is typically loaded on system start-up, and is used to configure each Bluetooth adapter attached to the system. The formatting, syntax, and usage of this file is well documented in the `hcid.conf` manpage.

7.2 `hcitool`

The tool `hcitool` has two main uses. The first is to search for and detect nearby Bluetooth devices, and the second is to test and show information about low-level Bluetooth connections. In a sense, `hcitool` picks up where `hciconfig` ends – once the Bluetooth adapter starts communicating with other Bluetooth devices.

Detecting nearby Bluetooth Devices

`hcitool scan` searches for nearby Bluetooth devices and displays their addresses and user-friendly names:

```
# hcitool scan
Scanning ...
        00:11:22:33:44:55       Cell Phone
        AA:BB:CC:DD:EE:FF       Computer-0
        01:23:45:67:89:AB       Laptop
        00:12:62:B0:7B:27       Nokia 6600
```

In this invocation, four Bluetooth devices were found. Detecting the addresses of nearby Bluetooth devices and looking up their user-friendly names are actually two separate processes, and conducting the name lookup can often take quite a long time. If the user-friendly name is not needed, then `hcitool inq` is useful for performing only the first part of the search, finding the addresses of nearby devices.

Testing Low-Level Bluetooth Connections

The tool `hcitool` can be used to create piconets of Bluetooth devices and show information about locally connected piconets. When writing Bluetooth software, we won't have to worry about these low-level details, just like we won't have to worry about instructing the Bluetooth adapter on which radio frequencies to use. For application programming, this part of `hcitool` is strictly of educational use, because BlueZ automatically takes care of piconet formation and configuration in the process of establishing higher level (RFCOMM) and (L2CAP) connections.

For the reader who is curious about using it for basic piconet configuration, the `hcitool cc` and `hcitool con` commands are the first places to start. `hcitool cc` forms a piconet with another device, and is fairly straightforward to use. For example, to join a piconet with the device 00:11:22:33:44:55,

```
# hcitool cc 00:11:22:33:44:55:66
```

`hcitool con` can then be used to show information about existing piconets.

```
# hcitool con
Connections:
        < ACL 00:11:22:33:44:55 handle 47 state 1 lm MASTER
```

Here, the output of `hcitool con` indicates that the local Bluetooth adapter is the master of one piconet, and the device 00:11:22:33:44:55 is a part of that piconet. For details on the rest of the output, see the `hcitool` documentation.

> *Note:* A fairly common mistake is to try to use `hcitool` to create data transport connections between two Bluetooth devices. It's important to know that even if two devices are part of the same piconet, a higher level connection needs to be established before any application-level data can be exchanged. Creating the piconet is only the first step in the communication process. When writing code, as shown in the preceding chapters, this is not needed.

7.3 `sdptool`

The tool `sdptool` has two uses. The first is for searching and browsing the Service Discovery Protocol (SDP) services advertised by nearby devices. This

is useful for seeing what Bluetooth profiles are implemented by another Bluetooth device, such as a cellular phone or a headset. The second is for basic configuration of the SDP services offered by the local machine.

Browsing and Searching for Services

`sdptool browse [addr]` retrieves a list of services offered by the Bluetooth device with address `addr`. Omitting `addr` causes `sdptool` to check all nearby devices. If `local` is used for the address, then the local SDP server is checked instead. Each service record found is then briefly described. A typical service record might look like this:

```
# sdptool browse 00:11:22:33:44:55
Browsing 00:11:22:33:44:55
Service Name: Bluetooth Serial Port
Service RecHandle: 0x10000
Service Class ID List:
  "Serial Port" (0x1101)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 1
Language Base Attr List:
  code_ISO639: 0x656e
  encoding:    0x6a
  base_offset: 0x100
Profile Descriptor List:
  "Serial Port" (0x1101)
    Version: 0x0100
```

Here, the device `00:11:22:33:44:55` is advertising a single service called "Bluetooth Serial Port" that's operating on RFCOMM channel 1. The service has the Universally Unique Identifier (UUID) 0x1101, and also adheres to the Bluetooth Serial Port Profile, as indicated by the profile descriptor listed at the bottom. In general, this information is sufficient for an application to determine whether or not this is the service that it's looking for (has UUID 0x1101), and how to connect to it (use RFCOMM channel 1).

`sdptool search` can be used to search nearby devices for a specific service. The service to search for can be specified either as a UUID or as a preset string

such as "SP" (Serial Port) or "OPUSH" (OBEX Object Push). To obtain a full list of the preset strings, invoke `sdptool -h`. For example, to search for services advertising the Serial Port UUID invoke

```
# sdptool search SP
```

Alternatively, to search directly on its UUID

```
# sdptool search 0x1101
```

Basic Service Configuration

`sdptool add <name>` can be used to advertise a set of predefined services, all of which are standardized Bluetooth Profiles. It cannot be used to advertise an arbitrary service with a user-defined UUID; this must be done programmatically. This means it won't be very useful for advertising a custom service.

`sdptool del <handle>` can be used to unadvertise a local service. The SDP server maintains a `handle` for each service that identifies it to the server, essentially a pointer to the service record. To find the handle, just look at the description of the service using `sdptool browse` and look for the line that says "Service RecHandle:". Using the example above, the Serial Port service has the handle `0x10000`, so on that machine, issue the following command to stop advertising the service:

```
# sdptool del 0x10000
```

`sdptool` also provides commands for modifying service records (e.g., to change a UUID), which you could actually use, but probably don't want to. These, along with the `add` and `del` commands, exist so that programmers can look at the source code of `sdptool` for examples on how to do the same in their own applications.

7.4 `hcidump`

For low-level debugging of connection setup and data transfer, `hcidump` can be used to intercept and display all Bluetooth packets sent and received by the local machine. This can be very useful for determining how and why a connection fails, and let us examine at exactly what stage in the connection process communications failed. `hcidump` requires superuser privileges.

When run without any arguments, `hcidump` displays summaries of Bluetooth packets exchanged between the local computer and the Bluetooth adapter as they appear. This includes packets on device configuration, device

inquiries, connection establishment, and raw data. Incoming packets are preceded with the ">" (greater than) symbol, and outgoing packets are preceded with the "<" (less than) symbol. The length of each packet (`plen`) is also shown. For example, `hcidump` started in one command shell and the command `hcitool inq` issued in another, the output of `hcidump` might look like this:

```
# hcidump
HCI sniffer – Bluetooth packet analyzer ver 1.23
device: hci0 snap_len: 1028 filter: 0xffffffff
< HCI Command: Inquiry (0x01|0x0001) plen 5
> HCI Event: Command Status (0x0f) plen 4
> HCI Event: Inquiry Result (0x02) plen 15
> HCI Event: Inquiry Complete (0x01) plen 1
```

Here, we can see that one command (Inquiry) was sent out instructing the Bluetooth adapter to search for nearby devices, and three packets of size 5, 4, and 15 bytes were received: information on the status of the command, an inquiry result indicating that a nearby device was detected, and another status packet once the inquiry completed. Notice that used this way, `hcidump` provides only basic summaries of the packets, which is not always enough for debugging. One option is to use the `-X` flag, which causes `hcidump` to display the raw contents of every packet in hexadecimal format along with their ASCII decodings. Used in the above example, we might see the following:

```
# hcidump -X
HCI sniffer – Bluetooth packet analyzer ver 1.23
device: hci0 snap_len: 1028 filter: 0xffffffff
< HCI Command: Inquiry (0x01|0x0001) plen 5
  0000: 33 8b 9e 08 00                                        3....
> HCI Event: Command Status (0x0f) plen 4
  0000: 00 01 01 04                                           ....
> HCI Event: Inquiry Result (0x02) plen 15
  0000: 01 26 75 05 3d 0f 00 01  02 00 00 01 3e d6 1f  .&u.=.......>..
> HCI Event: Inquiry Complete (0x01) plen 1
  0000: 00                                                    .
```

Okay, so unless you've memorized the Bluetooth specification and can decode the raw binary packets in your head, maybe that's not as useful as we'd like. While `hcidump -X` is great for very low level debugging of raw packets, the `-V` option is a nice compromise. `hcidump -V` displays as much information as it can gather from each packet, and summarizes the ones it can't interpret. If used together with `-X`, it still provides all the information for packets that

it can decode, but also shows the raw hexadecimal data for all the other packets. (These tend to be application-level data packets.) Repeating our example once again, we might see this:

```
# hcidump -X -V
HCI sniffer – Bluetooth packet analyzer ver 1.23
device: hci0 snap_len: 1028 filter: 0xffffffff
< HCI Command: Inquiry (0x01|0x0001) plen 5
    lap 0x9e8b33 len 8 num 0
> HCI Event: Command Status (0x0f) plen 4
    Inquiry (0x01|0x0001) status 0x00 ncmd 1
> HCI Event: Inquiry Result (0x02) plen 15
    bdaddr 00:0F:3D:05:75:26 mode 1 clkoffset 0x1fd5 class 0x3e0100
> HCI Event: Inquiry Complete (0x01) plen 1
    status 0x00
```

Now, the packets decoded according to the Bluetooth specification can be seen, which are probably mostly meaningless to you right now, but would make sense if you found the need to read the parts of the Bluetooth specification on device inquiry. Since this is a simple example, `hcidump` is able to fully decode each packet, so there is no raw hexadecimal data.

As with the other utilities, there are many more ways to use `hcidump` for debugging and low-level display of Bluetooth packet communication that you can find by reading the help text included with BlueZ.

7.5 `l2ping`

The tool `l2ping` sends echo packets to another Bluetooth device and waits for a response. An echo packet is a special type of L2CAP packet that contains no meaningful data; when a Bluetooth device receives an echo packet, it should just send (echo) the packet back to the originator. This is useful for testing and analyzing L2CAP communications with another Bluetooth device. If two devices are communicating, but seem a little sluggish, then `l2ping` can provide timing information on how long it takes to send and receive packets of a certain size. The only required parameter is the address of the Bluetooth device to "ping." For example, to send echo packets to the device `01:23:45:67:89:AB`:,

```
# l2ping -c 5 01:23:45:67:89:AB
Ping: 01:23:45:67:89:AB from 00:D0:F5:00:0E:B5 (data size 44) ...
```

```
44 bytes from 01:23:45:67:89:AB id 0 time 60.87ms
44 bytes from 01:23:45:67:89:AB id 1 time 55.97ms
44 bytes from 01:23:45:67:89:AB id 2 time 50.96ms
44 bytes from 01:23:45:67:89:AB id 3 time 51.94ms
44 bytes from 01:23:45:67:89:AB id 4 time 48.93ms
```

`l2ping` continues sending packets until stopped by pressing `Ctrl-C`. Other command-line arguments let us control the size of the packets sent, the delay between packets, how many to send, and so on. For details on how to use these capabilities, invoke `l2ping -h`.

7.6 `uuidgen`

As the name suggests, `uuidgen` generates UUIDs that, for all intents and purposes, can be considered unique from all other UUIDs. This is most useful when designing new Bluetooth applications that make use of SDP to advertise a nonstandard UUID (e.g., a new Service Class ID). To use this tool, simply invoke it and record the generated number. For example,

```
# uuidgen
30c007a7–7792–49e7–a266–60426e757f18
```

7.7 Summary

The GNU/Linux user-level tools for Bluetooth development are highlighted in this chapter because of their value in debugging and understanding a Bluetooth application. Since Bluetooth deals with multiple device communicating wirelessly, these tools are useful not only for debugging an application written for the GNU/Linux platform but also for verifying that applications written on other platforms appear and behave as expected. Often, an application will fail unexpectedly, or it may not operate as hoped with a new device. In the best case, a few simple steps, such as an `sdptool browse` to verify that a service record is properly advertised or an `hcitool scan` to verify that a device is discoverable, may resolve the problem. For more difficult bugs, `hcidump` can (and often will) be used to analyze the raw packets being transmitted over a Bluetooth connection to determine exactly where in the communications process the problem lies.

Index