Bluetooth for Programmers

Albert Huang

albert@csail.mit.edu

Larry Rudolph

rudolph@csail.mit.edu

Bluetooth for Programmers
by Albert Huang and Larry Rudolph

TODO

Table of Contents

List of Tables

Preface

1. About this book

There are loads and loads of material already out there about Bluetooth. The problem with all of them is that they just have too much information. Specifically, they try to tell *all about Bluetooth* when most of the time, we're only interested in a tiny fraction.

This book purposefully and happily leaves out a great deal of information about Bluetooth. A lot of concepts are simplified and described in ways that make sense, not necessarily the ways they're laid out in the Bluetooth specification. The key is that they're described in the simplest way possible so that you as a programmer can start working with those concepts.

This book is not meant to be a be-all-end-all guide to Bluetooth programming. Instead, it's meant to serve as a stepping stone, the first foothold on which programmers interested in working with Bluetooth can start from. Once you've read through and understood the concepts and techniques in this book, you'll have enough knowledge to start creating your own functional Bluetooth applications that can interoperate with many other Bluetooth devices. If you find yourself wanting to know more about the inner-workings and nitty-gritty details of Bluetooth, you'll also be well prepared to tackle the more complex and technical documents like the Bluetooth specification itself, which gives you enough information to build your own Bluetooth chip from scratch.

2. Audience

This book targets the computer programmer looking for an introduction to Bluetooth and how to program with it in Linux. 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 experience with either C or Python, and have access to and can use a Linux 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. It helps to have a basic understanding of these concepts as the comparisons will help solidify your understanding of Bluetooth programming.

3. Organization of This Book

TODO

4. Acknowledgments

TODO

4.1. Albert’s acknowledgments

TODO

4.2. Larry’s acknowledgments

TODO

Chapter 1. Introduction

In a single phrase, Bluetooth is *a way for devices to communicate with each other wirelessly over short distances.* A comprehensive set of documents, called the Bluetooth Specifications, describes in gory detail exactly how they accomplish this, but the basic idea is about wireless, short-range communication.

TODO

1.1. Understanding Bluetooth as a software developer

Developing applications that make use of Bluetooth communication is straightforward and easy although it may seem difficult due to its unusually wide scope. Technologies names or specifications, often refer to something very specific and with a narrow scope. Ethernet, for example, describes how to connect a bunch of machines together to form a simple network, but that's about it. TCP/IP describe 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. But if someone asked you to describe the Internet, where would you start? What would you explain? You might describe Ethernet, TCP/IP, email, or the World-Wide-Web, or all of them at once. The hard part is knowing where to start because there is so much to describe at so many different levels. On the other hand, if a software developer approached you and wanted to know about Internet programming - how to connect one computer on the Internet to the other and send data back and forth, you probably wouldn't bother describing the details of Ethernet or email, precisely because they are both technologies aren't central to answering that question. Sure, you might mention email as an example of what Internet programming can accomplish, or describe Ethernet to give context on how the connections are implemented, but what you'd really want to describe is TCP/IP programming.

In many ways, the word Bluetooth is like the word Internet because it encompasses a wide range of subjects. Similar to Ethernet or USB, Bluetooth defines a lot of physical on-the-wire stuff like on which radio frequencies to transmit 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.

Despite all that Bluetooth encompasses, a programmer only needs to know a small fraction of what's laid out in the specifications. When a software developer approaches to ask about how to get started with Bluetooth programming, you really only need to describe how to connect one Bluetooth device to another, and how to transfer data between the two. Sure, it helps to know a bit about the rest of Bluetooth, but there's no need to go into the specifics of the algorithms that Bluetooth devices use to choose on their radio frequencies. The bad news is that Bluetooth is more than just a replacement for a USB or ethernet cable. Most network application do not need to if their machine is connected tothe network via a physical ethernet cable or a wireless 802.11 connection, they do need to know if the connection is Bluetooth. The good news, is that they do not need to know very much.

1.2. Bluetooth Programming Concepts

The previous section gave a general overview of Bluetooth as a communications technology, and information that's useful to know about Bluetooth but isn't absolutely necessary to create functional programs. This section focuses specifically on explaining the parts of Bluetooth that concern a software developer. Throughout the rest of this chapter, we'll often present Bluetooth concepts side by side with concepts from Internet programming. This is in part because the vast majority of network programmers are already familiar with TCP/IP to some degreer. It is also because Bluetooth programming shares so much in common with Internet programming, and it makes sense to explain a new idea in terms of an old idea when they're not all that different.

Although Bluetooth was designed from the ground up, independently of the Ethernet and TCP/IP protocols, it is quite reasonable to think of Bluetooth programming in the same way as Internet programming. Fundamentally, they have the same principles of one device communicating and exchanging data with another device.

The different parts of network programming can be separated into several components

- Choosing a device with which to communicate
- Figuring out how to communicate with it
- Making an outgoing connection
- Accepting an incoming connection
- Sending and receiving data

Some of these components do not apply to all models of network programming. In a connectionless model, for example, there is no notion of establishing a connection. Some parts can be trivial in certain scenarios and quite complex in another. If the numerical IP address of a server is hard-coded into a client program, for example, then choosing a device is no choice at all. In other cases, the program may need to consult numerous lookup tables and perform several queries before it knows its final communication endpoint.

1.2.1. Choosing a communication partner

Every Bluetooth chip ever manufactured is imprinted with a globally unique 48-bit address, which we will refer to as the *Bluetooth address* or *device address*. This is identical in nature to the MAC addresses of Ethernet [1], and both address spaces are actually managed by the same organization - the IEEE Registration Authority. These addresses are assigned at manufacture time and 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. This address is used at 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 as their data link layer 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 to communicate with it.

In both cases, the client program will often not have advance knowledge of these target addresses. In Internet programming, the user will typically supply a host name, such as `www.kernel.org`, which the client must translate to a physical IP address using the Domain Name System (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.

1.2.1.1. Device Name

Since humans do not deal well with 48-bit numbers like `0x000EED3D1829` (in much the same way we do not deal well with numerical IP addresses like 64.233.161.104), Bluetooth devices will almost always have a user-friendly 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 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 5 different phones, each of which is named "My Phone".

Although names in Bluetooth differ from Internet names in that there is no central naming authority and names can sometimes be the same, the client program still has to translate from the user-friendly names presented by the user to the underlying numerical addresses. In TCP/IP, this involves contacting a local nameserver, issuing a query, and waiting for a result. In Bluetooth, where there are no nameservers, a client will instead broadcast inquiries to see what other devices are nearby and query each detected device for its user-friendly name. The client then chooses whichever device has a name that matches the one supplied by the user.

1.2.1.2. Searching for nearby devices

THIS SHOULD REALLY BE A SIDE NOTE

Device discovery, the process of searching for and detecting nearby Bluetooth devices is often a confusing and irritating subject for Bluetooth developers and users. Why’s that, you might ask? Well, the source of this aggravation stems from the fact that it can take a long time to detect nearby Bluetooth devices. To be specific, if you have a Bluetooth cell phone and a Bluetooth laptop sitting next to each other on your desk and you want your phone to make a connection to your laptop, it will usually take an average of 5 seconds before it detects your laptop, and sometimes as long as 10-15 seconds. This might not seem like that much time, but if you put it in context and realize that while it’s performing its search, the phone is changing frequencies more than a thousand times a second and there are only 79 possible frequencies [2] that it can transmit on, then you’d start to wonder why they don’t find each other in the blink of an eye. The technical reasons for this aren't very interesting, but it's mostly due to the result of a strangely designed search algorithm. Suffice to say, device discovery may often take much longer than you'd like it to.

1.2.2. Choosing a transport protocol

Once our client application has determined the address of the host machine it wants to connect to, it must determine which transport protocol to use. This section describes the Bluetooth transport protocols closest in nature to the most commonly used Internet protocols. Consideration is also given to how the programmer might choose which protocol to use based on the application requirements.

Both Bluetooth and Internet programming involve using numerous different transport protocols, some of which are stacked on top of others. In TCP/IP, many applications use either TCP or UDP, both of which rely on IP as an underlying transport. TCP provides a connection-oriented method of reliably sending data in streams, and UDP provides a thin wrapper around IP that unreliably sends individual datagrams of fixed maximum length. There are also protocols like RTP for applications such as voice and video communications that have strict timing and latency requirements.

While Bluetooth does not have exactly equivalent protocols, it does provide protocols which can often be used in the same contexts as some of the Internet protocols.

1.2.2.1. RFCOMM

The RFCOMM protocol provides roughly the same service and reliability guarantees as TCP. Although the specification explicitly 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), it is quite simple to use it in many of the same scenarios as TCP.

In general, applications that use TCP are concerned with having a point-to-point connection over which they can reliably exchange streams of data. If a portion of that data cannot be delivered within a fixed time limit, then the connection is terminated and an error is delivered. Along with its various serial port emulation properties that, for the most part, do not concern network programmers, RFCOMM provides the same major attributes of TCP.

The biggest difference between TCP and RFCOMM from a network programmer's perspective is the choice of port number. Whereas TCP supports up to 65535 open ports on a single machine, RFCOMM only allows for 30. This has a significant impact on how to choose port numbers for server applications, and is discussed shortly.

1.2.2.2. L2CAP

UDP is often used in situations where reliable delivery of every packet is not crucial, and sometimes to avoid the additional overhead incurred by TCP. Specifically, UDP is chosen for its best-effort, simple datagram semantics. These are the same criteria that L2CAP satisfies as a communications protocol.

L2CAP, by default, provides a connection-oriented [3] protocol that sends individual datagrams of fixed maximum length. The default maximum packet size is 672 bytes, but this can be negotiated up to 65535 bytes. Being fairly customizable, L2CAP can be configured for varying levels of reliability. To provide this service, the transport protocol that L2CAP is built on [4] employs an transmit/acknowledgement scheme, where unacknowledged packets are retransmitted. There are three policies an application can use:

- never retransmit
- retransmit until success or total connection failure (the default)
- drop a packet and move on to queued data if a packet hasn't been acknowledged after a specified time limit (0-1279 milliseconds). This is useful when data must be transmitted in a timely manner.

Never retransmitting and dropping packets after a timeout are often referred to as *best-effort* communications. Trying to deliver a packet until it has been acknowledged or the entire connection fails is known as *reliable* communications. Although Bluetooth does allow the application to use best-effort instead of reliable communication, several caveats are in order. The reason for this is that adjusting the delivery semantics for a single L2CAP connection to another device affects *all* L2CAP connections to that device. If a program adjusts the delivery semantics for an L2CAP connection to another device, it should take care to ensure that there are no other L2CAP connections to that device. Additionally, since RFCOMM uses L2CAP as a transport, all RFCOMM connections to that device are also affected. While this is not a problem if only one Bluetooth connection to that device is expected, it is possible to adversely affect other Bluetooth applications that also have open connections.

The limitations on relaxing the delivery semantics for L2CAP aside, it serves as a suitable transport protocol when the application doesn't need the overhead and streams-based nature of RFCOMM, and can be used in many of the same situations that UDP is used in.

Given this suite of protocols and different ways of having one device communicate with another, an application developer is faced with the choice of choosing which one to use. In doing so, we will typically consider the delivery reliability required and the manner in which the data is to be sent. As shown above and illustrated in Table 1-1, we will usually choose RFCOMM in situations where we would choose TCP, and L2CAP when we would choose UDP.

Table 1-1. A comparison of the requirements that would lead us to choose certain protocols. Best-effort streams communication is not shown because it reduces to best-effort datagram communication.

Requirement	Internet	Bluetooth
Reliable, streams-based	TCP	RFCOMM

Requirement	Internet	Bluetooth
Reliable, datagram	TCP	RFCOMM or L2CAP with infinite retransmit
Best-effort, datagram	UDP	L2CAP (0-1279 ms retransmit)

1.2.3. Port numbers and the Service Discovery Protocol

The second part of figuring out how to communicate with a remote machine, once a numerical address and transport protocol are known, is to choose the port number. Almost all Internet transport protocols in common usage are designed with the notion of port numbers, so that multiple applications on the same host may simultaneously utilize a transport protocol. Bluetooth is no exception, but uses slightly different terminology. In L2CAP, ports are called *Protocol Service Multiplexers*, and can take on odd-numbered values between 1 and 32767. Don't ask why they have to be odd-numbered values, because you won't get a convincing answer. In RFCOMM, *channels* 1-30 are available for use. These differences aside, both protocol service multiplexers and channels serve the exact same purpose that ports do in TCP/IP. L2CAP, unlike RFCOMM, has a range of reserved port numbers (1-1023) that are not to be used for custom applications and protocols. This information is summarized in Table 1-2. Throughout the rest of this book, we'll often use the word *port* instead of protocol service multiplexer and channel, mostly for clarity.

Table 1-2. Port numbers and their terminology for various protocols

protocol	terminology	reserved/well-known ports	dynamically assigned ports
TCP	port	1-1024	1025-65535
UDP	port	1-1024	1025-65535
RFCOMM	channel	none	1-30
L2CAP	PSM	odd numbered 1-4095	odd numbered 4097 - 32765

In Internet programming, server applications traditionally make use of well known port numbers that are chosen and agreed upon at design time. Client applications will use the same well known port number to connect to a server. 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.

The Bluetooth transport protocols, however, were designed with many fewer available port numbers, which means we cannot choose an arbitrary port number at design time. Although this problem is not as significant for L2CAP, which has around 15,000 unreserved port numbers, RFCOMM has only 30 different port numbers. A consequence of this is that there is a greater than 50% chance of port number collision with just 7 server applications. In this case, the application designer clearly should not arbitrarily choose port numbers. The Bluetooth answer to this problem is the Service Discovery Protocol (SDP).

Instead of agreeing upon a port to use at application design time, the Bluetooth approach is to assign ports at runtime and follow a publish-subscribe model. The host machine operates a server application, called the SDP server, that uses one of the few L2CAP reserved port numbers. Other server applications are dynamically assigned port numbers at runtime and register a description of themselves and the services they provide (along with the port numbers they are assigned) with the SDP server. Client applications will then query the SDP server (using the well defined port number) on a particular machine to obtain the information they need.

1.2.3.1. Service ID

This raises the question of how do clients know which service description is the one they are looking for. The easy answer would be to just assign every single service a unique identifier and be done with it. 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 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.

UUIDs are typically referred to as 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 8 digits corresponds to bits 1-32 of the UUID, the next segment of 4 digits is bits 33-36, and so on.

1.2.3.2. 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 only useful 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 also are 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 only have *one* Service ID, it can have many *Service Class IDs*.

NOTE: Technically, the Bluetooth specification demands that every SDP service record have an Service Class ID list with at least one entry. I think that's stupid. The Linux Bluetooth implementation does not enforce this. Should we mention this?

Bluetooth Reserved UUIDs

Similar to the way L2CAP and TCP have reserved port numbers for special purposes, Bluetooth also has reserved UUIDs. These are mostly used for identifying predefined service classes, but also for transport protocols and profiles (Bluetooth profiles are described in Section 1.3.5). Usually, you'll see them referred to as 16-bit or 32-bit values, but they do correspond to full 128-bit UUIDs.

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*

1.2.3.3. SDP attributes

So far, we've described SDP as a way to figure out what port and protocol a particular application service is running on, using a Service ID or a Service Class ID as a lookup key. A more general way to think of SDP is as an information database. Every record advertised by SDP is actually a list of *attributes*, where each attribute is in turn an *[ID, value]* pair. The attribute *ID* is a 16-bit unsigned integer that specifies the type of attribute, and the actual attribute data is described in the *value* field. A client application looking for a service can search on any of these attributes, although most will usually search on the two already mentioned.

The data in the *value* field is not restricted to only UUIDs, and can also be an integer, a boolean value, a text string, a list of any of those types, or even a list of lists. Attributes values can be of variable length - up to 4 GB long, although you'd have to be a little crazy to actually try that. All of this makes SDP a powerful way of describing services, but also makes it a bit complicated and sometimes tedious to work with.

Bluetooth defines several reserved attribute IDs which always have a special meaning, and the rest can be used any way an application designer wishes to. Some of the more common reserved attributes are:

Service class ID list

A list of service class UUIDs that the service provides.

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 1.3.5. Each descriptor consists of a UUID and a version number.

DIAGRAM!!!

1.2.3.4. Is SDP even necessary?

In this section, we've seen how to avoid the pitfalls of fixed port numbers and how a client program can use SDP to find the specific Bluetooth service it's looking for. Knowing this, we should also keep in mind that SDP is not even required to create a Bluetooth application. It is perfectly possible to revert to the TCP/IP way of assigning port numbers at design time and hope to avoid port conflicts, and this might often be done to save some time. In controlled settings such as the computer laboratory or an in-house project, this is quite reasonable. Ultimately, however, to create a portable application that will run in the greatest number of scenarios, the application should use dynamically assigned ports and SDP.

1.2.4. Communicating using sockets

It turns out that choosing which machine to connect to and how to connect are the most difficult parts of Bluetooth programming. Once the transport protocol and port number to cmomunicate on are chosen, the rest of Bluetooth communications is essentially the same type of programming most network programmers are already accustomed to: 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. In this section, we'll give a brief introduction to the concepts behind socket programming. Like the rest of this chapter, we won't distract you with any code yet, just give an overview of what's involved. If you're already a seasoned veteran with socket programming, then you can skip this section, but if you're new to sockets, then read on!

1.2.4.1. Introducing the Socket

DIAGRAM!!! 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 being 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.

To establish a Bluetooth connection, a program must first `create` a socket that will serve as the endpoint of the connection. Sockets are used for all types of network programming, so the first thing to do is specify what kind of socket it's going to be. In Bluetooth programming, we'll almost always be creating either L2CAP or RFCOMM sockets, so that all data sent over the sockets will be sent using the correct protocol.

When first created, the socket is not yet connected and can't be used yet for communication. To connect it, however, the application must decide if the socket will be used as a server socket to listen for incoming connections, or as a client socket to establish an outgoing connection. The process of connecting the socket depends on this choice, so we'll look at each case separately.

1.2.4.2. Client sockets

Client sockets are easy to understand and straightforward to use. Once the socket has been created, the client program only needs to issue the `connect` command, specifying which device to connect to, and on which port. 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.

1.2.4.3. Server / Listening sockets

To get a useful data connection out of a server socket (also called listening sockets), there are three steps an application must take. First, it must `bind` the socket to local Bluetooth resources, specifying which Bluetooth adapter and which port number to use[5]. Second, the socket must be placed into `listening` mode. This indicates to the operating system that it should listen for connection requests on the adapter and port number chosen during the bind step. Finally, the application uses the bound and listening socket to `accept` incoming connections.

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, what happens is each time the server socket accepts a new incoming connection, 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. DIAGRAM!!!

1.2.4.4. Communicating using a connected socket

Once a Bluetooth application has a connected socket, using it to communicate it simple. The `send` and `receive` commands are used to... well, send and receive data. 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.

1.2.4.5. Nonblocking sockets with `select`

TODO

1.2.4.6. Socket summary

To briefly summarize, socket programming is a multi-step process that involves 8 main operations.

Create

  Allocates an unconnected socket.

Connect (client)

  Establishes an outgoing connection. Implicitly forms a piconet if necessary.

Bind (server)

  Reserves a port number on a local Bluetooth adapter.

Listen (server)

  Instructs the operating system to begin accepting incoming connections.

Accept (server)

  Waits for incoming connections.

Receive

  Receive incoming data on a Bluetooth connection.

Send

  Send data to the remote device of a Bluetooth connection.

Close

  Disconnects a connected socket, or shuts down a listening socket.

1.3. Useful things to know about Bluetooth

One does not need to know very much about section

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 1-3 summarizes their differences. Almost all Bluetooth-enabled cell phones, headsets, laptops, and other consumer-level Bluetooth devices are class 2 devices. There are many class 1 USB devices for sale to consumers. It is the higher class that determines the properties. If a class 1 USB device communicates with a class 2 Bluetooth cell phone, the range of the Bluetooth radio is limitted by the cell phone. Class 3 Bluetooth device are rare, as their limited range heavily restricts their usefulness.

Table 1-3. The three Bluetooth power classes

Power class	Transmission power level	Advertised range
1	100 mW	100 meters
2	2.5 mW	10 meters
3	< 1 mW	< 1 meter

The ranges listed here are only rough estimates used for advertising purposes. In practice, one can see a much larger range when there aren’t many obstructions between two devices, and a smaller range when there’s a lot of radio interference or objects in between. People are actually quite good at blocking Bluetooth signals, mostly because water (which constitues around 60% of the human body) does a great job absorbing radio waves at the frequencies used by Bluetooth. Distance is only related to the transmission power. Further distances may have higher error rates and a device might be seen outside its low-error operating range.

1.3.2. Communications Speed

It is also difficult to give a reliable number 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 you’re likely to see will be a bit less since there’s always going to be a bit of 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 how far apart they are. Unless you can closely control the distance and obstructions between two Bluetooth devices, 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, which was released in late 2004, have a theoretical limit triple that of older devices (2178.1 kb/s asymmetric, 1306.9 kb/s symmetric), but at the time of this writing (October, 2005) there aren't very many Bluetooth 2.0 devices available on the market, and the vast majority of existing devices are limited by the older data rates.

1.3.3. Radio Frequencies and Channel Hopping

Bluetooth devices all operate in the 2.4 GHz frequency band. This means that it uses the same radio frequencies as microwaves, 802.11, and some cordless phones (the kind that attach to land lines, not cell 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.

DIAGRAM!! For comparison, take a look at the way 802.11b and 802.11g work. Both of these wireless networking technologies divide the 2.4 GHz band into 14 channels that are 5 MHz wide. When a wireless network is setup, 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, Bluetooth 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 microseconds (1600 times per second). It tries to do this in a fairly 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.

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 co-exist 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, the Bluetooth designers came up with some even more confusing terminology that you might hear a lot, but 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.

DIAGRAM!! 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 8 devices 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.

DIAGRAM!! To better understand the master device's second role, we'll compare it again with how 802.11 works. In 802.11, there is no such thing as an orderly way of transmitting. If a device has a data packet to send to another, it waits until no other device is transmitting, then waits a little more, and then transmits. If the recipient got the message, then it 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. 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 to get their voice heard, and a moderated discussion where the moderator gives everyone who raises their hands 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 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. Bluetooth Profiles + RFCs

Along with the simple TCP, IP, and UDP transport protocols used in Internet programming, there are a host of other protocols to specify, in great detail, methods to route data packets, exchange electronic mail, transfer files, load web pages, and more. Once standardized by the Internet Engineering Task Force in the form of Request For Comments (RFCs) [6], these protocols are generally adopted by the wider Internet community. Similarly, Bluetooth also has a method for proposing, ratifying, and standardizing protocols and specifications that are eventually adopted by the Bluetooth community. The Bluetooth equivalent of an RFC is a Bluetooth Profile.

Due to the short-range nature of Bluetooth, the Bluetooth Profiles tend to be complementary to the Internet RFCs, with emphasis on tasks that can assume physical proximity. For example, there is a profile for exchanging physical location information [7], a profile for printing to nearby printers [8], and a profile for using nearby modems [9] to make phone calls. There is even a specification for encapsulating TCP/IP traffic in a Bluetooth connection, which really does reduce Bluetooth programming to Internet programming.

If you find yourself needing to implement one of the Bluetooth Profiles, you can find the specification and all the details for that particular profile on the Bluetooth website http://www.bluetooth.org/spec, where they are freely distributed.

Notes

1. http://www.ietf.org/rfc/rfc0826.txt
2. The device discovery process actually only uses 24 of the 79 channels, which makes it even sillier
3. The L2CAP specification actually allows for both connectionless and connection-based channels, but connectionless channels are rarely used in practice. Since sending “connectionless" data to a device requires joining its piconet, a time consuming process that is merely establishing a connection at a lower level, connectionless L2CAP channels afford no advantages over connection-oriented channels.
4. Asynchronous Connection-Less logical transport
5. Most computers only have one Bluetooth adapter, so choosing a Bluetooth adapter isn’t much of a choice at all
6. http://www.ietf.org/rfc.html
7. Local Positioning Profile
8. Basic Printing Profile
9. Dial Up Networking Profile

Chapter 2. Bluetooth programming with Python - PyBluez

Now that we have an understanding of the concepts needed for Bluetooth programming, it’s time to get our hands dirty and learn how to implement each of those different parts. To do this, we’re going to use Python as a learning tool. Why Python, you might ask? Why not Java, or C, or (insert your favorite language here)? There are two answers to that question. The short answer is that it’s just plain easy, as we’ll soon find out. 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".

The only tricky part we have to deal with before getting started is making sure that we add Bluetooth support to Python. Although Python has a large and comprehensive standard library, Bluetooth is not yet part of the standard distribution. Enter PyBluez, a Python extension that provides Python programmers with access to system Bluetooth resources on GNU/Linux computers. Once we have this installed, as described in *TODO*, we’re ready to get up and running.

Note: If you’re not very comfortable with Python, don’t worry! The examples used in this chapter use only the simplest parts of Python possible, and you should think of reading through the examples as if you’re reading pseudocode.

2.1. Choosing a communication partner

Following the steps outlined in Chapter 1, the first action a Bluetooth program should take is to choose a communication partner. Example 2-1 shows a Python program that looks for a nearby device with the user-friendly name “My Phone". An explanation of the program follows.

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 partner 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?

PyBluez represents a Bluetooth address as a string of the form "xx:xx:xx:xx:xx", where each x is a hexadecimal character representing one byte of the 48-bit address, with the most significant byte listed first. Bluetooth devices in PyBluez will always be identified using an address string of this form. In the previous example, if the target device had address "01:23:45:67:89:AB", we might see the following output:

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

`discover_devices` is used in this example without any arguments, which should be sufficient for most situations, but there are a couple ways we can tweak it. 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 avoid these cached results, set the *`flush_cache`* parameter to `True`. We can also control the amount of time that `discover_devices` spends scanning with the *`duration`* parameter, which is specified in integer units of 1.28 seconds. This somewhat strange number is a consequence of the Bluetooth specification - device scans always last a multiple of *exactly* 1.28 seconds. It's usually not a good idea to decrease this below the default value of 8 (10.24 seconds).

`lookup_name` also takes a parameter that controls how long it spends searching. If `lookup_name` is not able to determine the user-friendly name of the specified Bluetooth device within a default value of 10 seconds, 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 to keep in mind 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

Example 2-2 and Example 2-3 show the basics of how to establish a connection using an RFCOMM socket, transfer some data, and disconnect. In the first example, a server application waits for and accepts a single connection on RFCOMM port 1, receives a bit of data and prints it on the screen. 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

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

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

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

A Bluetooth socket in PyBluez is represented as an instance of the `BluetoothSocket` class, and almost all communications will use methods of this class. 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.

An RFCOMM `BluetoothSocket` used to accept incoming connections must be attached to operating system resources with the `bind` method. `bind` takes in a single parameter - a tuple specifying the address of the local Bluetooth adapter to use and a port number to listen on. 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 `listen` puts the socket into listening mode and it is then ready to accept incoming connections with the `accept` method.

`accept` returns two values - a brand new `BluetoothSocket` object connected to the client, and the connection information as a *`address`*, *`port`* pair - *`address`* corresponds to the Bluetooth address of the connected client and *`port`* is the port number on the client's side of the connection.

Client programs do not need to call `bind` or the other two server-specific functions, but instead use the `connect` method to establish an outgoing connection. Like `bind`, `connect` also takes a tuple specifying an address and port number, but in this case the address can't be empty and must be a valid Bluetooth address. In Example 2-3 , the client tries to connect to the Bluetooth device with address "01:23:45:67:89:AB" on port 1. This example, and Example 2-2 , assumes that all communication happens on RFCOMM port 1. Section 2.4 shows how to dynamically choose ports and use SDP to search for which port a server is operating on.

Once a socket is connected, the `send` and `recv` methods can be used to, well... send and receive data. `recv` takes a parameter specifying the maximum amount of data to receive, specified in bytes, and returns the next data packet on the connection. To send a packet of data over a connection, simply pass it to `send`, which queues it up for delivery.

Once an application is finished with its Bluetooth communications, it can disconnect by calling the `close` method on a connected socket. So how does one side detect when the other has disconnected? The `recv` method will return an empty string. This is the only case where `recv` does that, which makes it a reliable way of knowing when the connection has been terminated.

We've left out error handling code in these examples for clarity, but the process is fairly straightforward. If 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

Example 2-4 and Example 2-5 demonstrate the basics of using L2CAP as a transport protocol. You'll notice that using L2CAP sockets is almost identical to using RFCOMM sockets.

Example 2-4. l2cap-server.py

```
from bluetooth import *

port = 0x1001

server_sock=BluetoothSocket( L2CAP )
server_sock.bind(("",port))
server_sock.listen(1)

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 *

sock=BluetoothSocket(L2CAP)

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

sock.connect((bd_addr, port))

sock.send("hello!!")

sock.close()
```

Aside from passing in `L2CAP` as a parameter to the `BluetoothSocket` constructor instead of `RFCOMM`, the only major difference between these examples and the RFCOMM examples from the previous section is the choice of port number. Remember that in L2CAP, we're strictly limited to odd-valued port numbers between 4097 and 32765. Usually, we'll use hexadecimal notation when referring to L2CAP port numbers, just because they tend to look a little cleaner.

2.3.1. Maximum Transmission Unit

As a datagram-based protocol, packets sent on L2CAP connections have an upper size limit. Although this has a small default value of 672 bytes, it can be adjusted. Each device at the endpoint 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 65535 bytes and as low as 48 bytes. In PyBluez, the `set_l2cap_mtu` function is used to adjust this value.

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

This method is fairly straightforward, and takes two parameters. *`l2cap_sock`* should be a connected L2CAP `BluetoothSocket`, and *`new_mtu`* is an integer specifying the incoming MTU for the local computer. 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 )
```

If you do find yourself using this function, don't forget that both devices involved in a connection should raise their MTU settings. It is possible for each side to have a different MTU, but that just gets confusing.

2.3.2. Best-effort transmission

Although we expressed reservations about using best-effort L2CAP channels in Section 1.2.2.2, there are some cases where we might prefer best-effort semantics over reliable semantics. For example, if we're 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( address, timeout )
```

`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 re-established, 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 device and establish the two main types of data transport connections, all using fixed Bluetooth address and port numbers that were determined at design time. To build a truly robust Bluetooth application service, we should use dynamically allocated port numbers. In doing so, we also need to give client applications a way to determine which port the service is running on. 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, Example 2-6 and Example 2-7 show the RFCOMM client and server from Section 2.2 modified to use dynamic port numbers and SDP. An explanation follows the examples.

Example 2-6. rfcomm-server-sdp.py

```
from bluetooth import *

port = get_available_port( RFCOMM )

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

advertise_service( server_sock, "Bluetooth Serial Port",
                   service_classes = [ SERIAL_PORT_CLASS ],
                   profiles = [ SERIAL_PORT_PROFILE ] )

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

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

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

Example 2-7. rfcomm-client-sdp.py

```
import sys
from bluetooth import *

service_matches = find_service( name = "Bluetooth Serial Port",
                                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("hello!!")
sock.close()
```

You’ll notice right away that these examples aren’t much different from the ones we saw in Section 2.2. Instead of hard-coding a port number, the server dynamically allocates a port number. After creating a bound and listening socket, the server then advertises an SDP service and continues on in the same manner as the previous examples. The client, instead of hardcoding a Bluetooth address and port number, searches for a service record and uses that information to establish a connection. In the next few pages, we’ll see some more details on how all this happens.

2.4.1. Dynamically allocating port numbers

Instead of using a predetermined port number, a Bluetooth server application can use the `get_available_port` function to find an unused port number.

```
    free_port = get_available_port( protocol )
```

This function takes a single parameter, *`protocol`*, which can be either `L2CAP` or `RFCOMM` and specifies which protocol the application will use. It checks each port starting from the lowest number and returns the first one that isn’t being used. The server application can then use `free_port` in a call to `bind`. If no ports are available at all, then it returns `None`.

`get_available_port` only identifies free ports, and doesn’t reserve them, so your application should make a call to `bind` immediately afterwards. It is possible that, in the few milliseconds of time between identifying the free port and binding it, another application could sneak by and "steal" the port number. If this happens, `bind` will raise a `BluetoothError`, so you can just repeat the process. This should almost never happen, but if you want to have a completely bug-free program that guards against this problem, you could do the following:

```
from bluetooth import *
socket = BluetoothSocket( RFCOMM )
while True:
    free_port = get_available_port( RFCOMM )
    try:
        socket.bind( ( "", free_port ) )
        break
    except BluetoothError:
        print "couldn’t bind to ", free_port

# listen, accept, and the rest of the program...
```

2.4.2. 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="", descrption="" )
```

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.

Calling `advertise_service` will register 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 can `stop_advertising`.

2.4.3. Searching for and browsing services

To find a single service, or get a listing of services on one or multiple nearby Bluetooth devices, we use the function `find_service`.

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

Without any arguments at all, `find_service` returns a listing of all services offered by all nearby Bluetooth devices. If there are a lot of Bluetooth devices in range, this could take a long time! Three optional parameters to this function can be used to filter the search results:

`name`

Optional. Restricts search results to services with this name. In the special case that this is `"localhost"`, then the local SDP server is searched.

`uuid`

Optional. Restricts search results to services with any attribute value matching this *`uuid`*. Note that the matching UUID could be either the service ID, or an entry in the service class ID list, or an entry in the profiles list.

`address`

Optional. Only searches the Bluetooth device with this *`address`*.

The results of this search is a list of dictionary objects. Each dictionary has eight keys, which describe the corresponding service. The value for a key may be `None`, which indicates that it wasn't specified in the service record. The keys and their values 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-classses"`

A list of service class IDs, in the same format as used for `advertise_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 the `_bluetooth` module.

2.5.1. Asynchronous socket programming with `select`

In the communications routines described so far, there is usually some sort of waiting involved. During this time, the controlling thread blocks and can't do anything else, such as respond to user input or display progress information. To avoid these pitfalls of *synchronous* programming, it is possible to use multiple threads of control, with one thread dedicated to each task that requires some waiting. That can get quite hairy and complicated, though, so instead we'll turn to using *asynchronous* techniques as a solution.

The first step in asynchronous programming is to switch the sockets to *non-blocking* mode, so that all the operations that would block (wait) beforehand return immediately instead. The idea is "Don't wait for something to happen. Just get it started and we'll figure it out later". To switch a socket into non-blocking mode, use the `setblocking` method and pass it `False`. Conversely, to switch back into blocking mode, pass it `True`. For example:

```
from bluetooth import *
sock = BluetoothSocket( RFCOMM )
sock.setblocking( False )
s.bind(("", get_available_port( RFCOMM )))
# ...
```

The `setblocking` method must be called on every socket that you want to switch to nonblocking mode. This includes sockets that are returned by the `accept` method.

The next step in asynchronous programming is the "Figure it out" step, where the program determines if anything happened. The idea here is to consolidate all of the things a program can wait on into one place. Then, when anything happens, some data is received or the user types something or a timer fires, the program can deal with it immediately. To do this, we can use the `select` module, which comes as part of the standard Python distribution. Within the `select` module is the `select` function, which is what we'll be using extensively.

```
from select import *

can_rd, can_wr, has_exc = select( to_read, to_write, to_exc, [timeout] )
```

`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 more 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.

So what exactly are the different types of events? Some of these should be pretty obvious, but others have been shoehorned in. Table 2-1 summarizes which list to put a socket in for detecting specific events.

Table 2-1. `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

You'll notice a couple 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 non-blocking operations, but hopefully that will change in the future.

2.5.2. Asynchronous device discovery

Asynchrously 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 2-8 shows an example of how to use `select` and `DeviceDiscoverer` for this purpose.

Example 2-8. 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
```

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.

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 https://www.bluetooth.org/foundry/assignnumb/document/baseband. The `DeviceDiscoverer` class can be used directly with the `select` module.

2.5.3. The `_bluetooth` module

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 almost cases without having to resort to the C API.

2.5.3.1. 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 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 millseconds 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, and it would be too boring to list them here. Needless to say, examples do help, so TODO

Example 2-9. Reading the user-friendly name of a local Bluetooth adapter

```
TODO
```

Chapter 3. C programming with `libbluetooth`

There are reasons to prefer developing Bluetooth applications 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 standalone application. As of this writing, BlueZ is a powerful Bluetooth communications stack with extensive APIs that allows a user to fully exploit all local Bluetooth resources, but it has no official documentation. Furthermore, there is very little unofficial documentation as well. Novice developers requesting documentation on the official mailing lists [1] are typically rebuffed and told to figure out the API by reading through the BlueZ source code. This is a time consuming process that can only reveal small pieces of information at a time, and is quite often enough of an obstacle to deter many potential developers.

This chapter presents a short introduction to developing Bluetooth applications in C with BlueZ. The tasks covered in chapter 2 are now explained in greater detail here for C programmers.

3.1. Choosing a communication partner

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 *ii = NULL;
    int max_rsp, num_rsp;
    int dev_id, sock, len, flags;
    int i;
    char addr[19] = { 0 };
    char name[248] = { 0 };

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

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

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

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

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

3.1.1. Compiling the example

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

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

3.1.2. 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 the `bdaddr_t`, which is simply a packed array of six bytes. 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 );
```

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

3.1.3. Choosing 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.

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 dev_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.

3.1.4. Scanning for nearby devices

After choosing the local Bluetooth adapter to use and allocating system resources, the program is ready to scan for nearby Bluetooth devices. In the example, `hci_inquiry` performs a Bluetooth device discovery and returns a list of detected devices and some basic information about them in the variable `ii`.

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

Here, the function doesn't actually use the socket opened in the previous step. Instead, `hci_inquiry` takes the resource number returned by `hci_get_route` (or `hci_devid`) as its first parameter. Most other functions we'll see will use the socket opened by `hci_open_dev`, but this one creates its own internal socket. The inquiry lasts for at most 1.28 * *`len`* seconds, and at most *`max_rsp`* devices will be returned in the output parameter *`ii`*, which must be large enough to accommodate *`max_rsp`* results. We suggest using a *`max_rsp`* of 255 for a standard 10.24 second inquiry.

If *`flags`* is set to `IREQ_CACHE_FLUSH`, then the cache of previously detected devices is flushed before performing the current inquiry. Otherwise, if *`flags`* is set to 0, then the results of previous inquiries may be returned, even if the devices aren't in range anymore.

The `inquiry_info` structure is defined as

```
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;
```

For the most part, only the first entry - the `bdaddr` field, which gives the address of the detected device - is of any use. Occasionally, there may be a use for the `dev_class` field, which gives information about the type of device detected (i.e. if it's a printer, phone, desktop computer, etc.) and is described in the Bluetooth Assigned Numbers [2]. The rest of the fields are used for low level communication, and are not useful for most purposes. If you're interested, the Bluetooth specification has all the gory details.

3.1.5. Determining the user-friendly name of a nearby device

Once a list of nearby Bluetooth devices and their addresses has been found, the program determines the user-friendly names associated with those addresses and presents them to the user. 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)
```

`hci_read_remote_name` tries for at most *`timeout`* milliseconds to use the socket *`hci_sock`* to query the user-friendly name of the device with Bluetooth address *`addr`*. On success, *`hci_read_remote_name`* returns 0 and copies at most the first *`len`* bytes of the device's user-friendly name into *`name`*.

`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 use-rfriendly names of nearby Bluetooth devices.

3.1.6. Error handling

So far, all the functions introduced return an integer on completion. If the function succeeds in doing whatever it was the program requested, then the return value is always greater than or equal to 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 all of the socket functions described in the next few sections.

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 boils down to the same socket programming techniques introduced in Section 1.2.4, which are also widely used in Internet programming. To get us started, Example 3-2 and Example 3-3 show how to establish a connection using an RFCOMM socket, transfer some data, and disconnect. For simplicity, the client is hard-coded to connect to `01:23:45:67:89:AB`.

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;
    int opt = sizeof(rem_addr);

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

    // bind socket to port 1 of the first available
    // local bluetooth adapter
    loc_addr.rc_family = AF_BLUETOOTH;
    loc_addr.rc_bdaddr = *BDADDR_ANY;
    loc_addr.rc_channel = (uint8_t) 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 = read(client, buf, sizeof(buf));
    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 = (uint8_t) 1;
    str2ba( dest, &addr.rc_bdaddr );

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

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

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

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

Those who read through the previous chapter will notice that the examples have the same flow and structure used by the corresponding Python examples in Section 2.2. Additionally, the seasoned Internet programmer will notice that these two examples are almost exactly the same as corresponding examples used in TCP programming. The primary differences are in the way the sockets are created, and the addressing structures used. First, the `socket` function is used to allocate 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, `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 which is used as a handle to that socket.

3.2.1. Addressing structures

To establish an RFCOMM connection with another Bluetooth device, incoming or outgoing, create and fill out a `struct sockaddr_rc` addressing structure. Like the `struct sockaddr_in` that is used in TCP/IP, 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, `rc_bdaddr` and `rc_channel` specify the Bluetooth address and port number to connect to, 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.

3.2.2. Establishing a connection

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

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 );
```

`sock` should be 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. *`addrlen`* should always be `sizeof( struct sockaddr_rc )`.

Next, the application takes the bound socket and puts it into listening mode with 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, the new connection is put into a backlog queue. The *`backlog`* parameter specifies how big this queue should be. Usually, a value of 1 or 2 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.

3.2.3. Using a connected socket

Once 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 parameter, 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. Similarly, `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 in the addressing structure and the availability of a few more options to control. Example 3-4 and Example 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".

Example 3-4. l2cap-server.c

```
#include <stdio.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;
    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);

    memset(buf, 0, sizeof(buf));

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

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

Example 3-5. l2cap-client.c

```
#include <stdio.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 addr = { 0 };
    int s, status;
    char *message = "hello!";
    char dest[18] = "01:23:45:67:89:AB";

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

    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( status == 0 ) {
        status = write(s, "hello!", 6);
    }

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

    close(s);
}
```

For simple usage scenarios, the primary differences betweeen using RFCOMM sockets and L2CAP sockets are the parameters to the `connect` function, and the addressing structure used. 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.

For `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.

3.3.1. Byte ordering

Since Bluetooth deals with the transfer of data from one machine to another, the use of a consistent byte ordering for multi-byte 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 intenal 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. The first place we'll find a use for it is in specifying the port number in the `struct sockaddr_l2` structure. We didn't need it for the RFCOMM addressing structure because RFCOMM port numbers can be represented using a single byte, but representing an L2CAP port number requires two bytes. 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.

3.3.2. 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;
    };

    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, a program should first use `getsockopt` to retrieve the existing L2CAP options for a connected socket. After modifying the options, `setsockopt` should be used to apply the changes. For example, a function to do all of this might look like this:

```
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 programitcally 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.

3.4.1. 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 4097-32767 (0x1001 - 0x7FFF) instead of ports 1-30.

For Linux kernel versions 2.6.7 and greater, 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.

3.4.2. SDP data structures

Working with SDP in C can be a bit laborious because it requires using a few more data structures to represent 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 needed. If you're the type that likes to dive straight into examples, you may want to skip ahead to the next section and come back to 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 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 nodde.

The `sdp_list_append` function is used both for adding nodes to a list, and for 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 you're finished with a list, free the memory used by the list with the `sdp_list_free` function. When freeing a list, you can pass it a pointer to another function, which will be called on every data element in the list. The idea is that you can create a custom function to free the data elements, or use an existing function like `free`. If you don't pass in a function, and leave *`f`* set to `NULL`, then `sdp_list_free` does not modify or deallocate the data elements.

`sdp_profile_desc_t`

The `sdp_profile_desc_t` is used only when describing the Bluetooth profile that a service record adheres to.

```
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 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.

3.4.3. Advertising a service

Advertising a service can be broken up into two steps. The first step consists of building the service record that will be advertised, and the second step involves connecting to the local SDP server and actually registering the service. Building the service record can take up a fair amount of code, mostly because of the awkward way that data structures are handled in C, but everything after that is pretty simple. Example 3-6 shows a helper function that builds the service record and registers it with the local SDP server. It 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 <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_port = 11;
    const char *service_name = "Roto-Rooter Data Router";
    const char *service_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;

    // PART ONE

    // 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, service_dsc);

    // PART TWO

    // connect to the local SDP server, register the service record, and
    // disconnect
    session = sdp_connect( BDADDR_ANY, BDADDR_LOCAL, 0 );
    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 );

    return session;
}


int main()
{
    sdp_session_t* session = register_service();
    // The rest of the program here
    sdp_close( session );
    return 0;
}
```

After declaring a whole mess of local variables that will be used to store the different data elements of the service record, we start off 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 );
```

There are no reserved Service IDs in Bluetooth, so we always specify it as a full 128-bit number. Conveniently, a program can store the Service ID as an array of four 32-bit integers before converting it to the `uuid_t` data type, since that array takes up exactly 128-bits of memory. Then, pass the newly created `uuid_t` to `sdp_set_service_id`, which fills in the appropriate field of the service record *`rec`*, also passed in as a parameter.

Once the Service ID is done, move on 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 we encounter the `sdp_list_t`, which is used to store the list of UUIDs. `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, we create a list of `sdp_profile_desc_t` data structures, which are described earlier. `sdp_set_profile_descs` can then be used to set this list in the service record.

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

By now, you should have gotten the general idea of how to fill in a service record data structure. 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, 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.

`sdp_set_access_protos` is used to set which transport protocols are advertised in the service record, and is also where the port number being used by the server application gets defined. This one 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 lists contain 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.

`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`.

3.4.4. 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. You might say, "well why isn't there way to broadcast service searches?" and to that, I would say, "Good question!". Despite Bluetooth's piconet abilities, there is no way for a device to (metaphorically) shout out, "Does anyone have a printer!? Anyone?? A/S/L??" 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."

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. 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 value, so searching for that UUID is equivalent to asking for all services on a device.

Example 3-7. Step one of searching a device for a service with UUID 0xABCD

```
#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;
        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 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 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. Often, your program will be searching for a standard reserved UUID. In those cases, you can use the `sdp_uuid16_create` or `sdp_uuid32_create` functions described earlier in the chapter. If your program needs to search on more than one UUID at a time, then 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 progrram 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 should be a pointer to the `sdp_session_t` created above. *`uuid_list`* should be the list of UUIDs just created, and *`attrid_list`* should be 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 *`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 completed 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. If you read the previous section on advertising a service, these are the same `sdp_record_t` data structures that were created by the server application. 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 out of an `sdp_record_t` involves a number of helper functions. Typically, you won't access the data structure directly, but will instead use functions named *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 which port a service is running on, pass a `sdp_record_t` from the search results into `sdp_get_access_protos` along with the address of a `NULL` pointer. *`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 is 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

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 Host Controller Interface (HCI) socket, which provides a direct connection to the microcontroller on the local Bluetooth adapter. This socket type, introduced in section 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 Bluetooth Core Specification describes communication with a Bluetooth microcontroller in great detail, which we summarize here. The host computer can send commands to the microcontroller, and the microcontroller generates events to indicate command responses and other status changes. A command consists of a Opcode Group Field that specifies the general category the command falls into, an Opcode Command Field that specifies the actual command, and a series of command parameters. In BlueZ, `hci_send_cmd` is used to transmit a command to the microcontroller.

```
int hci_send_cmd(int sock, uint16_t ogf, uint16_t ocf, uint8_t plen,
                 void *param);
```

Here, `sock` is an open HCI socket, `ogf` is the Opcode Group Field, `ocf` is the Opcode Command Field, and `plen` specifies the length of the command parameters `param`.

Calling `read` on an open HCI socket waits for and receives the next event from the microcontroller. An event consists of a header field specifying the event type, and the event parameters. A program that requires asynchronous device detection would, for example, send a command with `ocf` of `OCF_INQUIRY` and wait for events of type `EVT_INQUIRY_RESULT` and `EVT_INQUIRY_COMPLETE`. The specific codes to use for each command and event are defined in the specifications and in the BlueZ source code.

3.5.1. Best-effort transmission

TODO

It is slightly misleading to say that L2CAP sockets are reliable by default. Multiple L2CAP and RFCOMM connections between two devices are actually logical connections multiplexed on a single, lower level connection [3] established between them. The only way to adjust delivery semantics is to adjust them for the lower level connection, which in turn affects *all* L2CAP and RFCOMM connections between the two devices.

As we delve deeper into the more complex aspects of Bluetooth programming, the interface becomes a little harder to manage. Unfortunately, BlueZ does not provide an easy way to change the packet timeout for a connection. 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 3-8 shows how to do this.

Example 3-8. 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 status = 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 ) {
        status = dd;
        goto cleanup;
    }
    status = ioctl(dd, HCIGETCONNINFO, (unsigned long) cr );
    if( status != 0 ) 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
    status = hci_send_req( dd, &rq, 0 );
    if( status != 0 ) goto cleanup;

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

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

On success, the packet timeout for the low level connection to the specified device is set to `timeout * 0.625` milliseconds. A timeout of 0 is used to indicate infinity, and is how to revert back to a reliable connection. The bulk of this function 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`. In most cases, BlueZ provides convenience functions to construct the packets, send them, and wait for the response. Setting the packet timeout, however, seems to be so rarely used that no convenience function for it currently exists.

3.6. Chapter Summary

This chapter has provided an introduction to Bluetooth programming with BlueZ. The concepts covered in chapter 2 were presented here in greater detail with examples on how to implement them in BlueZ. Many other useful aspects of BlueZ were left out for brevity. Specifically, the command line tools and utilities that are distributed with BlueZ, such as `hciconfig`, `hcitool`, `sdptool`, and `hcidump`, are not described here. These utilities, which are invaluable to a serious Bluetooth developer, are already well documented. Only the simplest aspects of using the Service Discovery Protocol were covered - just enough to search for and advertise services. Additionally, other socket types such as `BTPROTO_SCO` and `BTPROTO_BNEP` were left out, as they are not crucial to forming a working knowledge of programming with BlueZ. Unfortunately, as of now there is no official API reference to refer to, so more curious readers are advised to download and examine the BlueZ source code [4].

Notes

1. http://www.bluez.org/lists.html (http://www.bluez.org/lists.html)
2. https://www.bluetooth.org/foundry/assignnumb/document/baseband
3. Bluetooth terminology refers to this as the ACL connection
4. available at http://www.bluez.org

Chapter 4. Bluetooth development tools

Note: need to re-word this introduction now that the chapter is after 2 and 3

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 daemons. Roughly speaking, the kernel part is responsible for managing the Bluetooth hardware resources that are attached to a machine, wrestling with all the different types of bluetooth adapters that are out there, 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 takes the interface exposed by the Linux kernel and 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.

The user-level tools are the programs that a typical end-user or programmer might use to leverage the computer's Bluetooth capabilities, while the daemons are constantly running programs that use the Bluetooth development library to manage the system's Bluetooth resources in the ways configured by the user. 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, you'll be interacting with the user-level tools the most, so we'll focus on introducing them in this chapter.

There are six command-line tools provided with BlueZ that are indispensable when configuring Bluetooth on a machine and degugging applications. We'll give some short descriptions here on how they're useful, and show some examples on how to use them. For full information on how to use them, you should consult the `man` pages that are distributed with the tools, or invoke each tool with the `-h` flag. This section serves mainly to give you an idea of what the tools are and which one to use for what scenario.

4.1. `hciconfig`

`hciconfig` is used to configure the basic properties of Bluetooth adapters. 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, and so on). Most of the commands require superuser privileges. Some of the most useful ways to use this tool are:

Display the status of recognized Bluetooth adapters

```
# 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 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 can be used to enabled and disable a Bluetooth adapter. To check whether or not a device is enabled, use `hciconfig` without any arguments.

```
# 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’
```

"Hide" an adapter, or show it to the world.

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. Don't be confused by the names! These control whether the adapter *responds* to inquiries and to pages (connection requests), not whether it makes them.[1]

Table 4-1. Inquiry Scan and Page Scan

Inquiry Scan	Page Scan	Interpretation	command
On	On	This is the default. The adapter is detectable by other Bluetooth devices, and will accept incoming connection requests	`piscan`
Off	On	Although not detectable by other Bluetooth devices, the adapter still accepts incoming connection requests by devices that already know the Bluetooth address of the adapter.	`pscan`
On	Off	The adapter is detectable by other Bluetooth devices, but it wil not accept any incoming connections. This is mostly useless.	`iscan`
Off	Off	The adapter is not detectable by other Bluetooth devices, and will not accept any incoming connections.	`noscan`

For example, the following invocation disables both Inquiry Scan and Page Scan for the first Bluetooth adapter.

```
# hciconfig hci0 noscan
```

There are many more ways to use `hciconfig`, all of which are described in the help text (`hciconfig -h`) and the man pages (`man hciconfig`). The key thing to remember is that `hciconfig` is the tool to use for any non-connection related settings for a Bluetooth adapter.

NOTE: Changes made by `hciconfig` are only temporary, and the effects are erased after a reboot or when the device is disabled and enabled again. `hcid.conf` should be used To make a change permanent (e.g. to permanently change the user-friendly name).

NOTE: The name `hciconfig` comes from the term Host Controller Interface (HCI). It refers to the protocol that a computer uses to communicate with the Bluetooth microcontroller that resides on the computer's Bluetooth adapter. HCI is used to do all the dirty work of configuring the adapter and setting up connections. The commands `hciconfig` and `hcitool` are so named to emphasize that they are used for the low-level Bluetooth operations that, while important, can't actually be used for communicating with other Bluetooth devices.

4.2. `hcitool`

`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 fuond. 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 you don't need the user-friendly names, then `hcitool inq` is useful for only performing the first part of the search - finding the addresses of nearby devices.

Testing low-level Bluetooth connections

`hcitool` can be used to create piconets of Bluetooth devices and show information about locally connected piconets. Remember that piconets are just an ugly consequence of Bluetooth's fancy frequency hopping techniques. When we're 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. So 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.

If you're curious about using `hcitool` for basic piconet configuration, then 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` tells us 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 communications process.

4.3. `sdptool`

`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`. Leaving `addr` out 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 UUID 0x1101, and also adheres to the Bluetooth Serial Port Profile, as indicated by the profile descriptor list at the bottom. In general, this information should be 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, but it can only look for a handful of predefined services. It is not able to search for a service with an arbitrary UUID, this must be done programmatically. Because of this, `sdptool browse` will generally be more useful for testing and debugging applications that use SDP (e.g. to check that a service is being advertised correctly).

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 programatically. This means it won't be very useful for advertising a custom service.

`sdptool del <handle>` can be used to un-advertise 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 if we were using that machine, we could 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), that you could actually use, but probably don't want to. These, along with the `add` and `del` commands exist more so that programmers can look at the source code of `sdptool` for examples on how to do the same in their own applications. Advertising and configuring services with C and Python are described in later chapters of this book, but you can always download the BlueZ source code at http://www.bluez.org and see how it's done with `sdptool`.

4.4. `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 lets us examine at exactly what stage in the connection process did communications fail. `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 symobl. The length of each packet (`plen`) is also shown. For example, if we started `hcidump` in one command shell and issued the command `hcitool inq` 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. You'll notice that used this way, `hcidump` only provides 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 gives us a nice compromise. `hcidump -V` will display as much information as it can gather from each packet, and summarize the ones it can't interpret. If used together with `-X`, it will still provide all the information for packets that it can decode, but will also show 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, we see the packets decoded according to the Bluetooth specification, 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 we don't see any 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 out by reading the help text included with BlueZ.

4.5. `l2ping`

`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`.

4.6. `rfcomm`

The `rfcomm` tool lets us establish arbitrary RFCOMM connections and treat them like serial ports. Although the RFCOMM protocol was described in the previous chapter as a general purpose transport protocol, one of its original purposes was to emulate a serial port connection between two devices. The idea was that device manufacturers who had serial-port capable devices would only need to add a Bluetooth chip to the end of the serial port controller, which requires much less modification to the original device than replacing the serial port controller. In fact, Bluetooth was even marketed as a "wireless serial cable". To utilize the serial-port emulation capabilities of Bluetooth in Linux, we use the `rfcomm` tool.

`rfcomm` can be used to connect to another device or to listen for incoming connections. A special device file is created for each connection, which user-level programs can read and write to like regular files. Data written to the device file is transmitted over Bluetooth, and reading from the device file retrieves the data received over the connection. When the device file is closed, the Bluetooth connection is terminated.

To listen for an incoming connection, we first choose which device file to bind it to. Typically, we'll use `/dev/rfcommX`, where `X` ranges from 0 - 9. Next, we choose an RFCOMM port number to listen on. To listen on RFCOMM port 20 and connect it to `/dev/rfcomm0`, we'd use the `rfcomm listen` command like this:

```
# rfcomm listen /dev/rfcomm0 20
```

Similarly, to establish an outgoing connection and serial port, we'd use the `rfcomm connect` command, but we would also specify the address of the Bluetooth device to connect to:

```
# rfcomm connect /dev/rfcomm0 01:23:45:67:89:AB 20
```

Keep in mind that in both these examples, the special device file `/dev/rfcomm0` is not a valid file until the `rfcomm` commands successfully complete. The other way of using `rfcomm` to establish outgoing connections is to use the `rfcomm bind` command to create the device file, and only establish the Bluetooth connection when a separate program tries to access the device file. For example:

```
# rfcomm bind /dev/rfcomm0 01:23:45:67:89:AB 20
```

Using `rfcomm` in this way is sort of saying "When a program opens /dev/rfcomm0, make a connection to the Bluetooth device 01:23:45:67:89:AB and send all data through that file. But if no program ever access that file, don't bother making the connection"

4.7. `uuidgen`

TODO

4.8. Obtaining BlueZ and PyBluez

Note: this should be an appendix

Instructions for installing the BlueZ development libraries can be found at the BlueZ website: htp://www.bluez.org (http://www.bluez.org). Most modern Linux distributions should have this packaged somehow. For example, on Debian-based systems:

```
apt-get install libbluetooth1-dev bluez-utils
```

On Fedora:

```
yum install bluez-devel
```

Similarly, instructions for installing PyBluez can be found at the PyBluez website: http://org.csail.mit.edu/pybluez. PyBluez is included with a few Linux distributions, but TODO

Notes

1. The idea is that Inquiry Scan and Page Scan control whether the adapter *scans* for inquiries and pages, in the same way that you might use your eyes to scan around to see if anyone is talking to you. Confusing!

Chapter 5. Other platforms and programming languages

5.1. Microsoft Windows

TODO

introduce the Widcomm Bluetooth stack and development environment. Also mention the Bluetooth stack that Microsoft built into Windows XP with Serice Pack 1.

5.2. OS X

TODO

5.3. Symbian OS / Nokia Series 60 Smartphones

TODO

5.4. Java

There are a number of Java bindings for Bluetooth programming currently available. The Java community has the advantage of having standardized on an API for Bluetooth development, called JSR-82. Almost all Java Bluetooth implementations adhere to this specification. This makes porting Bluetooth applications from one device to another much simpler. Current implementations of JSR-82 for the GNU/Linux operating system include Rocosoft Impronto [1], Avetana [2], and JavaBluetooth [3].

A disadvantage of using Java is that JSR-82 is very limited, providing virtually no control over the device discovery process or established data connections. For example, JSR-82 provides no method for adjusting delivery semantics, flushing a cache of previously detected devices during a device discovery, or obtaining signal strength information [4]. While JSR-82 is acceptable for creating simple Bluetooth applications, it is not well suited for research and academic purposes. Furthermore, Java and many JSR-82 implementations are not available on a number of platforms.

TODO

Notes

1. http://www.rocosoft.com (http://www.rocococosoft.com)
2. http://www.avetana-gmbh.de/avetana-gmbh/produkte/jsr82.eng.xm (http://www.avetana-gmbh.de/avetana-gmbh/produkte/jsr82.eng.xm)
3. http://www.javabluetooth.org
4. Cache flushing and signal strength were not covered in this chapter, but are described in the PyBluez documentation and examples