Table 7.9
The event HCI_PDU.

Field	size	comments
HCI_Event_Header		
Event_Code	1 byte	identifies the event • '0xFE' is reserved for Bluetooth logo specific events • '0xFF' is reserved for vendor-specific events used during module manufacturing, such as module testing and debugging operations
Payload_Length	1 byte	length of the payload of the event HCI_PDU in bytes
HCI_Event_Payload		
Payload	*Payload _Length* bytes	the payload of an event HCI_PDU is structured as a sequence of variable-size fields for the various parameters related to this event

A host uses the command HCI_PDUs for things like:

- setting operational parameters of the module, such as providing a link key for authentication;
- configuring the module's operational status and related parameters, for instance causing it to activate and set the related parameters for a low power mode;
- reading and writing register entries, like the number of broadcast packet repetitions, N_{BC}, and so on.

Depending upon the command, module registers will be read or set, the link manager will execute an LMP transaction, the link controller will change state and execute, say, a page, and so on. The host controller notifies the host of the outcome of the command with an event HCI_PDU either soon after the command is sent from the host or at a later time when appropriate–for example, following the termination of an LMP transaction. The reason that host controller HCI_PDU transmissions to the host are called events and not responses is that the host controller may initiate its own request (for instance, requesting a missing link key from the host) or send a transmission to the host without the host's prior action (perhaps notifying it of a connection request coming from a remote device). Actually, some of the command HCI_PDUs sent from the host are simply responses to event HCI_PDUs that originated from the host

IPR2020-00202
Apple Inc. EX1057 Page 167

controller. For example, the *HCI_Accept_Connection_Request* command is sent by the host to the host controller instructing the latter to accept an incoming connection request from a remote device. Before the host transmits the *HCI_Accept_Connection_Request* command, the host controller notifies the host of the incoming connection request with a Connection_Request event.

Table 7.10 shows the structure of a data HCI_PDU.

Table 7.10
The data HCI_PDU.

field	size	comments
HCI_Data_Header		
Connection_Handle	12 bits	identifies the baseband link over which these data are transmitted or received; connection handles in the range '0xF00' to '0xFFF' are reserved for future use
Flags	4 bits	*ACL transmissions*: composed of two subfields: • *Packet_Boundary_Flag*: identifies the beginning or continuation of an upper-layer (L2CAP) PDU • *Broadcast_Flag*: identifies the "spread factor" for the ACL transmission: point-to-point, broadcast to active slaves, or broadcast to all slaves including any parked ones
		SCO transmissions: reserved field
Payload_Length	2 bytes	length of the payload of the data HCI_PDU in bytes
HCI_Data_Payload		
Payload	*Payload_Length* bytes	data to be carried over the ACL or SCO baseband link identified by the contents of the *Connection_Handle* field

Transmission of data HCI_PDUs across the physical interface is regulated by the buffer sizes available on the receiving side of the PDU. Both the host and the host controller inquire about the buffer size available for receiving data HCI_PDUs on the opposite side of the interface and adjust their transmissions accordingly. This implies that a large L2CAP_PDU may need to be fragmented within the HCI layer prior to

IPR2020-00202
Apple Inc. EX1057 Page 168

sending it to the host controller. On the receiving side, the HCI layer could reconstruct L2CAP_PDUs based on the packet boundary flag information within the received data HCI_PDUs. Transmission of HCI_PDUs across the physical interface is in first-in-first-out order without preemption. Commands are processed by the host controller in their order of arrival, but they may complete out of order since each might take a different amount of time to execute. Similarly, events are processed by the host in order of arrival, but their processing may terminate out of order.

Note that none of the fields in any of the HCI_PDUs identifies the HCI_PDU class: command, event or data. Identification of the HCI_PDU class is left to the HCI transport protocol that actually carries the PDUs between the host and the host controller. Strictly speaking, this is a violation of protocol layering. However, it allows the HCI to take advantage of the capabilities of the underlying transport protocol, which may provide its own means for distinguishing the three HCI_PDU classes with minimal overhead. Purists may wish to consider that the HCI layer in the host and its complementary part in the host controller consist of a transport-independent sublayer, and a transport dependent convergence sublayer (which executes the HCI transport protocol) that adapts the HCI_PDUs to the particular transport method used to carry them across the physical interface.

The HCI_PDUs

There are many command HCI_PDUs organized into several groups identified by the OGF subfield in the header of the command HCI_PDU. For many of these command HCI_PDUs there exists a corresponding event HCI_PDU that carries the outcome and return parameters related to the command. For several commands, information related to their status and execution results is carried by two special events: *Command_Status_Event* and *Command_Complete_Event*. The former typically is sent immediately after a command is received by the host controller to indicate the status of the command, such as command pending execution, command not understood, and so on. This provides a sort of acknowledgement of the command along with an indication of its processing status. The latter is used to indicate the completion of execution of a command and to return related parameters, including whether or not the requested command was executed successfully. Observe that multiple events might be generated in response to a single command.

IPR2020-00202
Apple Inc. EX1057 Page 169

There are command HCI_PDUs related to link controller actions, policy-setting commands, the host controller itself, and many others. Command and event HCI_PDUs number over 100; some of these are highlighted in the following sections. These selected HCI_PDUs are illustrative of the type of information and the level of detail that is communicated between the host and the host controller. For the full set of HCI_PDUs, refer to the specification.

Link Control HCI_PDUs

The commands in this group are identified via the OGF subfield with the value 'b000001'. This group contains commands that allow inquiries to be sent to discover other devices in the vicinity. There are commands to create and terminate ACL and SCO connections and to accept or reject incoming connection requests. There are commands for initiating authentication and encryption procedures as well as for transporting authentication keys and PINs from the host to the link controller. There are information commands in this group to request the user-friendly name of the remote device, the link manager options that it supports and the clock offset registered in the remote device.

Following are some examples of HCI_PDUs in this group. The *HCI_Inquiry* command PDU instructs the module to enter the inquiry mode, using a given inquiry access code, for a specified amount of time or until a specified number of responses is collected. This command is summarized in Table 7.11.

Table 7.11
The *HCI_Inquiry* command HCI_PDU.

Command_Name	***HCI_Inquiry***		
OCF	'b0000000001'		
Parameters	LAP	3 bytes	lower address part used for generating the inquiry access code
	Inquiry_Length	1 byte	indicates the maximum duration for this inquiry: 1.28 sec - 61.44 sec
	Num_Responses	1 byte	indicates the maximum number of responses to be collected

The inquiry mode originated by this command terminates either when *Inquiry_Length* time has elapsed or when the number of responding devices reaches *Num_Responses*, whichever occurs first.

The host controller returns information collected from inquiries to the host with the *Inquiry_Result_Event*[8] summarized in Table 7.12; the parameters of the event are derived from the FHS BB_PDUs (detailed in the previous chapter) that are received from the devices responding to the inquiries. A brief description of the parameters below is given in Table 7.13, which presents the command that uses these parameters.

Table 7.12
The *Inquiry_Result_Event* event HCI_PDU; the index *i* identifies each of the *Num_Responses* responding devices.

Event_Name	*Inquiry_Result_Event*	
Event_Code	'0x02'	
Parameters	*Num_Responses*	1 byte
	BD_ADDR[*i*]	6*Num_Responses bytes
	Page_Scan_Repetition_Mode[*i*]	1*Num_Responses byte(s)
	Page_Scan_Period_Mode[*i*]	1*Num_Responses byte(s)
	Page_Scan_Mode[*i*]	1*Num_Responses byte(s)
	Class_of_Device[*i*]	3*Num_Responses bytes
	Clock_Offset[*i*]	2*Num_Responses bytes

The *HCI_Create_Connection* command PDU instructs the module to create a connection with a specified device, using a given set of BB_PDU types for the ACL link. Since the connection process requires that the "local" device page the "remote" device, this command also provides information used to accelerate the paging process. The paging information becomes available to the host of a local device via an earlier *Inquiry_Result_Event* PDU, shown in Table 7.12. The *HCI_Create_Connection* command is summarized in Table 7.13.

8. This event is actually called "Inquiry Result" event. Once again we take liberty in the naming convention for consistency purposes with other PDUs.

IPR2020-00202
Apple Inc. EX1057 Page 171

Table 7.13
The *HCI_Create_Connection* command HCI_PDU.

Command_Name	*HCI_Create_Connection*		
OCF	'b0000000101'		
Parameters	*BD_ADDR*	6 bytes	identifies the remote device with which to establish a connection
	Packet_Type	2 bytes	indicates which BB_PDU types can be used by the link manager for the ACL link
	Page_Scan_Repetition_Mode	1 byte	indicates the page scan repetition mode, that is, how frequently the remote device enters the page scan mode, last reported by the remote device
	Page_Scan_Mode	1 byte	indicates the page scan mode supported by the device
	Clock_Offset	2 bytes	indicates the difference between the slave and master clocks, as calculated in the last communication between them
	Allow_Role_Switch	1 byte	indicates whether this (the paging) device will be the master or will allow the paged device to become the master if requested[1]

1. Master-slave role switching is described in the previous chapter.

Upon successful creation of the connection, a *Connection_Complete_Event* is sent to the hosts on both sides of the connection. The events contain the *Connection_Handles* for identifying the connection. The connection handles are assigned by each host controller independently and their scope is limited to the local device only.

Link Policy HCI_PDUs

The commands in this group are identified via the OGF subfield with the value 'b000010'. This group contains commands that allow a device to set a power-management policy through the hold, sniff, and park

baseband modes and to define the parameters for those modes. Also, there are commands that pass QoS parameters from the L2CAP layer to the link manager, learn about the role (master or slave) that the device[9] plays for a particular connection and request a role switch if needed.

Table 7.14 summarizes the HCI_PDU command that requests the host controller to instruct the link manager and the baseband to enter hold mode with the parameters provided. Similar commands exist for sniff and park modes.

Table 7.14
The *HCI_Hold_Mode* command PDU.

Command_Name	*HCI_Hold_Mode*		
OCF	'b0000000001'		
Parameters	*Connection_Handle*	2 bytes	identifies the connection (actually the ACL link) to be placed in hold mode; only the 12 LSBs are used
	Hold_Mode_Max_Interval	2 bytes	indicates the maximum negotiable hold interval (0.625 msec – 40.9 sec)
	Hold_Mode_Min_Interval	2 bytes	indicates the minimum negotiable hold interval (0.625 msec – 40.9 sec)

The host controller notifies the host when hold mode is entered or is terminated using the *Mode_Change_Event.*

Host Controller and Baseband HCI_PDUs

The commands in this group are identified via the OGF subfield with the value 'b000011'. This group contains commands that allow the host to access and configure various hardware registers that maintain operational parameters. Among the operations that can be performed are determining the types of events that the host controller can generate; reading, writing, and deleting stored keys; reading and writing the user-

9. Recall that information regarding the role that a device plays in a particular connection does not propagate through the stack beyond the link manager layer. A host needs to explicitly request this information from the host controller.

friendly device name; activating and deactivating inquiry and/or page scans; reading and writing the authentication and/or encryption activity flag for a link; reading and writing the inquiry access codes used to listen during inquiry scans; forcing the flushing of ACL packets for a connection handle; reading and writing the audio codec parameters and so on.

Table 7.15 summarizes the HCI_PDU command that sets the inquiry scan parameters; a similar command exists for page scans as well. Inquiry scans occur only when the host has already sent an *HCI_Write_Scan_Enable* command PDU to enable inquiry scans.

Table 7.15
The *HCI_Write_Page_Scan_Activity* command PDU.

Command_Name	*HCI_Write_Inquiry_Scan_Activity*		
OCF	'b0000011100'		
Parameters	*Inquiry_Scan_Interval*	2 bytes	determines the interval between successive starts of inquiry scans 11.25 msec – 2.56 sec (typically, 1.28 sec)
	Inquiry_Scan_Window	2 bytes	determines the duration of a single continuous scan operation 11.25 msec – 2.56 sec (typically, 11.25 msec)

When the host controller finishes updating the related registers it returns a *Command_Complete_Event* to the host.

Informational Parameters HCI_PDUs

The commands in this group are identified via the OGF subfield with the value 'b000100'. This group includes commands that request static information about the hardware and firmware that is electronically "engraved" on the device at manufacture time. There is a command to request the version of the various protocols (LMP, HCI, and so on) that the module supports; a command to request a list of features supported by the link manager; a command to request the country of operation of the module; a command to request the *BD_ADDR* of the module;[10] and a command to request the host controller buffer information for ACL

10. Recall that the *BD_ADDR* is hardwired and cannot be modified.

IPR2020-00202
Apple Inc. EX1057 Page 174

and SCO packets, used to execute effective flow control at the host. The requested information is returned in a *Command_Complete_Event.*

Status Parameters HCI_PDUs

The commands in this group are identified via the OGF subfield with the value 'b000101'. This group includes commands that request information that is dynamically updated, like the value of the contact counter that measures the number of successive instants during which the remote device does not respond to local transmissions, causing the local link manager to flush any PDUs waiting to be transmitted. There is also a command HCI_PDU to retrieve information related to the quality of the link and the RSSI (received signal strength indicator) value. The requested information is returned in a *Command_Complete_Event.*

Testing HCI_PDUs

The commands in this group are identified via the OGF subfield with the value 'b000110'. These commands, which all result in *Command_Complete_Event* events, are used for testing the Bluetooth module and are not discussed further here.

Summary

In this chapter we have highlighted the upper two Bluetooth transport protocols: L2CAP and HCI. The latter is a transport protocol internal to a device, rather than an over-the-air protocol as are L2CAP and the other protocols discussed in Part 2 of the book. The primary purposes of these protocols are both to hide, in the case of L2CAP, and to expose, in the case of HCI, the internal operation of the lower transport protocols. L2CAP is used to multiplex and transport higher protocol layers while shielding them from the peculiarities of the lower transport protocols, like the baseband. The HCI provides a standardized interface to the services and capabilities provided by the lower transport protocols.

In this and the previous chapters we have presented the protocols that the SIG has developed for transporting data across Bluetooth devices. These protocols have been developed entirely by the SIG specifically for Bluetooth wireless communication. They reflect the SIG's objectives to develop simple, cost-effective communication systems that can operate at low power in noise-susceptible places. In the next chapter we introduce the middleware protocols that are used to take advan-

IPR2020-00202
Apple Inc. EX1057 Page 175

tage of the data-transport services of the transport protocols to enable a plethora of applications, including legacy applications, to operate smoothly over Bluetooth links.

8

The RFCOMM and SDP Middleware Protocols

We now move from the transport protocol layers to a detailed discussion of the middleware protocols. In this chapter we discuss RFCOMM, the Bluetooth serial port emulation protocol, and the Bluetooth Service Discovery Protocol, or SDP. Version 1.0 of the core specification (volume 1) devotes nearly 90 pages to these two protocols. As with the other detailed discussions of portions of the specification, this chapter attempts to reveal the motivation and thought process behind the development of these protocols. While the important elements of RFCOMM and SDP are examined here, this material focuses on the design basis for the protocols and thus is not a substitute for the specification itself.

Both RFCOMM and SDP reside directly above the L2CAP layer (discussed in the previous chapter) and use L2CAP connections to accomplish their respective functions. Both of these protocols provide a *protocol data unit* (PDU) structure for use by higher layers (either applications or other middleware protocols) in the stack. PDUs allow the higher layers of the stack to work with logical data elements at a higher level of abstraction than that of the packet format used by the transport protocols. Both RFCOMM and SDP are protocols developed specifically for use with Bluetooth wireless communications, although RFCOMM borrows heavily from an existing standard. Figure 8.1 illustrates the position of RFCOMM and SDP in the protocol stack. As shown in the figure, RFCOMM is used by higher layer middleware protocols and by applications for networking, IrDA interoperability and telephony. These same applications may communicate directly with

IPR2020-00202
Apple Inc. EX1057 Page 177

RFCOMM as well as with their associated middleware protocols that in turn communicate with RFCOMM. Since service discovery is fundamental to all Bluetooth profiles, most applications will also communicate with the SDP layer.

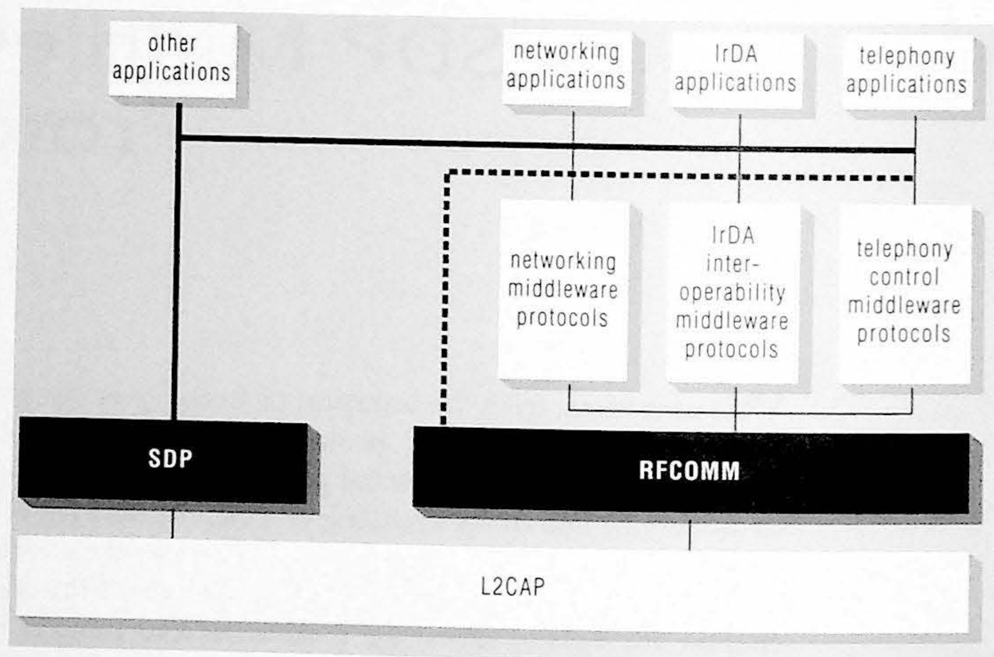

Figure 8.1
RFCOMM and SDP in the Bluetooth protocol stack.

The RFCOMM Protocol

Serial interfaces are ubiquitous in computing and telecommunications devices, particularly those devices with a high affinity for Bluetooth communications. Notebook computers have serial ports, personal digital assistants typically have serial ports (often used to synchronize the PDA with some other device), many mobile telephones have serial ports (often used for a wired headset), many digital cameras use serial ports to transfer image data to another device, printers and other computer peripherals often use serial ports for communication, and so on. Moreover, infrared communication, which as previously established has some traits in common with Bluetooth wireless communication, normally uses a serial port to communicate with the IR transceiver.[1]

Because Bluetooth technology aims to replace cables, it seems clear that there is a large opportunity to replace serial cables. To do this effectively, the stack needs to support serial communication in the same

manner as is done with cables, so that applications are presented with a familiar serial interface. This permits the cornucopia of legacy applications that are unaware of the Bluetooth technology to operate seamlessly over Bluetooth links. Furthermore, application software developers skilled in developing serial communication applications may still continue to do so, assured that their applications will operate over Bluetooth links. But the transport-layer protocols are not modeled after a serial port. L2CAP supports packet data structures, and while the air-interface may transmit bit patterns in a serial fashion, this is not the same as the common RS-232 types of serial interfaces used today with serial cables.

Thus the SIG has chosen to define a layer in the protocol stack that looks very much like a typical serial interface: the RFCOMM layer. In the world of personal computers, serial interfaces are often called *COM ports.* The name RFCOMM connotes a wireless (RF) instance of a virtual COM port. RFCOMM primarily is intended to enable cable-replacement scenarios for existing applications.

RFCOMM Protocol Development

The motivation for the RFCOMM protocol layer is rooted in the requirement to support legacy applications with initial Bluetooth implementations. The need for this serial communication function in the software stack was identified quite early in the SIG's formation. Just one month after the SIG was publicly announced, discussions ensued on developing the specification for the RFCOMM layer. At that time, the ETSI TS 07.10 standard [ETSI99] had already been identified as a candidate for a basis upon which to build Bluetooth serial communications. Requirements for Bluetooth serial communications include:

Multiplexed serial communications: There may be many simultaneous clients of the serial interface in the stack, including IrOBEX, telephony control and networking clients. Thus the serial port needs to be shareable through multiplexed connections (which

1. In the PC domain, infrared communications are frequently tied to a COM port resource. In commonly used PC operating environments, these COM ports classically have been difficult to configure, especially for infrared communications. This drawback has led to a situation where, while many infrared ports are deployed in products, only a fraction of these ports are actually used, since many users lack the expertise or motivation to perform the necessary configuration process. The rise of infrared ports on PDAs and mobile phones, where the configuration process is much easier, seems to lead to a higher usage rate of infrared in peer-to-peer communications.

IPR2020-00202
Apple Inc. EX1057 Page 179

in turn might be supported by the protocol multiplexing in the L2CAP layer, over which distinct RFCOMM entities in different devices communicate).

RS-232 signal compatibility: RS-232 is a widely used serial interface for the cables with which Bluetooth wireless communication needs to be compatible. Many applications are familiar with RS-232 interfaces, including the various control signals associated with RS-232; these include common signals such as Request to Send/Clear to Send (RTS/CTS), Data Terminal Ready/Data Set Ready (DTR/DSR), the RS-232 break frame and others. Emulating these signals allows RFCOMM to present to its clients the appearance of a serial port that is virtually the same as that used with a serial cable.

Remote status and configuration: In a peer-to-peer environment, the two parties communicating over the serial link often need to determine the status and configuration of the remote serial interface so that the local serial interface can be configured in a compatible manner. The service discovery protocol, discussed in following sections, can be used to obtain basic information needed to establish a serial communications channel; following connection establishment the two ends of the serial interface need to be able to negotiate compatible serial communication settings for the connection.

Internal and external serial port: To support the various uses of serial communications in Bluetooth applications, RFCOMM needs to support both an internal emulated serial port, in which the serial port parameters are used only locally (the parameters do not apply across the air-interface) as well as an external serial port, where the serial port parameters and status are transmitted across the RF link along with the data and may be used by the receiving serial port.

These requirements are not unique to Bluetooth environments, and the SIG realized that the aforementioned ETSI TS 07.10 standard was a good match for the needs of Bluetooth serial communications, so the SIG adopted much of that standard. However, TS 07.10 is not a perfect match for use in the Bluetooth protocol stack, so the SIG added some of its own modifications to adapt the ETSI standard for use in Bluetooth wireless communications. Among these additions and changes are:

Data frame adaptations: Since Bluetooth communication has an underlying packet structure by virtue of the use of L2CAP, some of

IPR2020-00202
Apple Inc. EX1057 Page 180

the data frame contents specified by TS 07.10 are unnecessary for RFCOMM. For example, the frame delimiter flags specified in TS 07.10 are discarded for the RFCOMM specification.

Connection establishment and termination: Again, because Bluetooth communication has its own connection management in the transport protocol layers that RFCOMM uses, the connection management functions of TS 07.10 are superfluous for RFCOMM. The specification details how RFCOMM links are managed.

Multiplexing: RFCOMM uses a subset of the multiplex channels specified for TS 07.10 and specifies the way in which some TS 07.10 multiplexing control commands are used in RFCOMM.

Applicability: The RFCOMM specification mandates support of several features which are optional in the TS 07.10 standard. These features deal with exchanging information about the configuration and status of the serial connection and include negotiating the serial port and individual channel settings and retrieving the serial port status. In Bluetooth environments these functions are quite useful and in fact can be considered necessary for effective use of the air-interface; thus they are specified as mandatory to support in RFCOMM.

Flow control: Typical serial ports pace the data transfer using XON/XOFF pacing or DTR signal pacing. For RFCOMM, the specification describes flow control mechanisms specific to the Bluetooth protocol stack, including flow control that applies to all multiplexed RFCOMM channels as well as per-channel pacing.

The remaining RFCOMM sections in this chapter review key points of the RFCOMM specification, in many cases highlighting the significance of the design choices for this protocol layer.

The RFCOMM Protocol Examined

The relatively few pages[2] devoted to RFCOMM in the specification belie its importance in the version 1.0 protocol stack. RFCOMM is the basis for most of the version 1.0 profiles and might also be used in some future profiles, although its primary purpose is to enable support for legacy applications in simple cable-replacement scenarios.[3] The main reason that RFCOMM does not require many dozens of pages of

2. Only about 25 pages, making the RFCOMM portion of the specification a good candidate for beginning-to-end reading for those interested in fully understanding this key layer of the stack.

IPR2020-00202
Apple Inc. EX1057 Page 181

explanation is the SIG's decision to adopt much of the ETSI TS 07.10 protocol (which itself is over 50 pages of specification). By specifying TS 07.10 as the basis for RFCOMM, the SIG has adopted a mature standard protocol and the specification needs to describe only the adaptation of this standard for Bluetooth environments. Much of the RFCOMM chapter of the specification focuses on describing which parts of TS 07.10 are relevant for RFCOMM, how those features are used and the modifications necessary to map TS 07.10 into the Bluetooth protocol stack.

RFCOMM uses an L2CAP connection to instantiate a logical serial link between two devices. In particular, an L2CAP connection-oriented channel is established that connects the two RFCOMM entities in the two devices. Only a single RFCOMM connection is permitted between two devices at a given time, but that connection may be multiplexed so that there can be multiple logical serial links between the devices.[4] The first RFCOMM client establishes the RFCOMM connection over L2CAP; additional users of the existing connection can use the multiplexing capabilities of RFCOMM to establish new channels over the existing link; and the last user to drop the final RFCOMM serial link should terminate the RFCOMM connection (and hence the underlying L2CAP connection). Each multiplexed link is identified by a number called a *Data Link Connection Identifier*, or DLCI. Figure 8.2 depicts multiplexed serial communications links using RFCOMM over L2CAP. In the illustration the various clients of RFCOMM each see their own emulated serial port, distinguished by a DLCI value (depicted by the different line types in the figure). These separate channels are then multiplexed over the RFCOMM link, which in turn is carried over an L2CAP connection.

3. RFCOMM might become less significant in future usage models as the specification evolves to support general TCP/IP networking. In the meantime, the SIG specified RFCOMM as a solution for serial-cable-replacement usage models.
4. Multiple links might be attained either through multiple instances of a single-channel RFCOMM or through a single instance of a multiple-channel RFCOMM (the latter being what the Bluetooth specification defines). While these might be logically equivalent, they are likely to result in real differences in implementations. The RFCOMM specification indicates that a client which requires a serial connection should first check for an already existing RFCOMM connection to the target device; if an RFCOMM connection to that device already exists, the client should just establish a new channel on that existing connection.

IPR2020-00202
Apple Inc. EX1057 Page 182

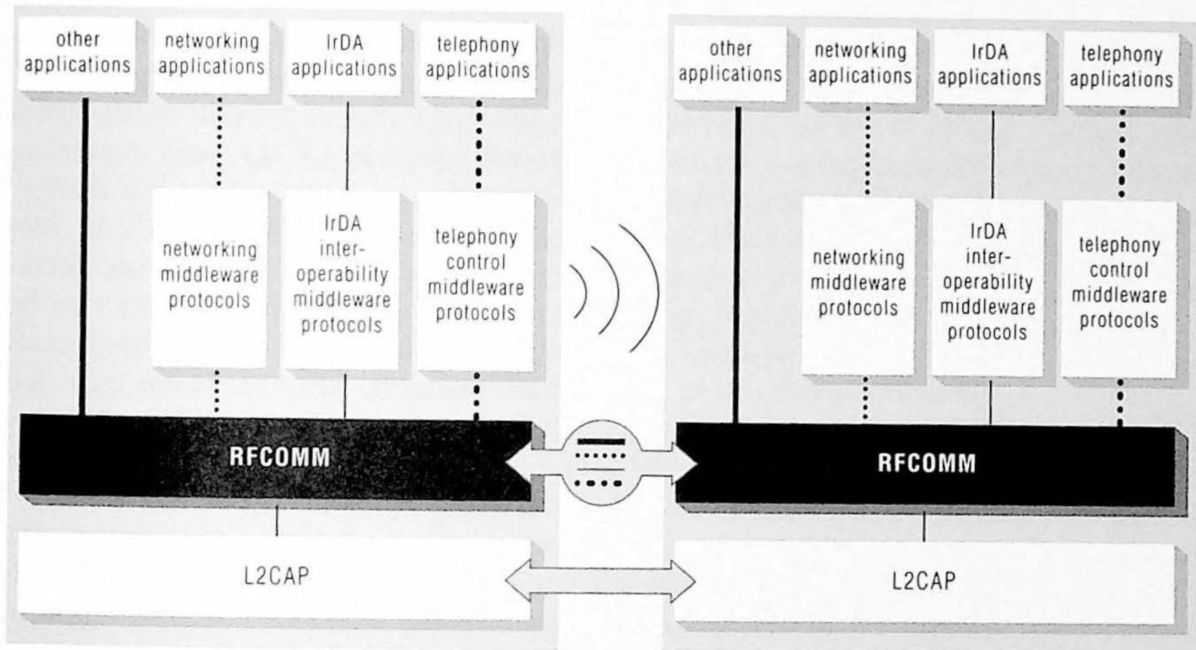

Figure 8.2
Multiplexed RFCOMM logical serial links (indicated by different line types) over a single RFCOMM connection, in turn over an L2CAP connection.

The specification allows for up to 60 multiplexed logical serial links over a single RFCOMM connection but does not mandate this level of multiplexing for RFCOMM implementations. In fact for most portable devices it is uncommon to have cases in which dozens of simultaneous serial links would be required in Bluetooth environments. Most devices are expected to support a fixed number of profiles, which will be a determining factor in how the protocol stack for those devices is implemented, including design tradeoffs such as the number of RFCOMM serial links supported. But consider also devices such as network access points that allow portable devices to use Bluetooth wireless communication to access larger networks (such as the Internet). The LAN Access profile (discussed in Chapter 15) specifies the use of PPP over RFCOMM, so a LAN access point device might indeed need many simultaneous serial connections to multiple devices. The RFCOMM specification supports this sort of usage by allowing more than one multiplexer session (that is, more than one instance of RFCOMM, in which case the multiplexing is achieved by using L2CAP's multiplexing capabilities), although such a capability is not mandated.

IPR2020-00202
Apple Inc. EX1057 Page 183

The RFCOMM chapter of the specification includes a discussion of two different sorts of devices that RFCOMM supports: *communication endpoint* (computer- or peripheral-style devices) and *communication mid-point* (modem-style devices). In general serial communications, these are often referred to as data terminal equipment (DTE) and data communications equipment (DCE), respectively. After making this distinction, though, the discussion concludes by stating that RFCOMM does not distinguish between these device types at all. In fact, it is not necessary for RFCOMM to do so; much as a standard serial cable can be configured for a direct serial connection or for null modem operation, RFCOMM also can be used in both manners. RFCOMM has included features to support DCE (modem-style) communications; these features may not be applicable for DTE communications. RFCOMM supports both device styles without needing to distinguish between them.

Typical cabled serial connections have a number of signals in the cable (usually nine for RS-232 communications, although all nine signals are not necessarily used in all applications). Bluetooth wireless communication obviously has no such signals because the transmission medium is the air-interface rather than a cable. In a multiplexed environment such as is defined by TS 07.10, it is desirable that each serial channel be viewed as an independent entity, with its own set of control and data signals. So even in a cabled environment some scheme is needed for multiplexing the serial signals. TS 07.10, and thus RFCOMM, do this by defining a specified control channel across which information is transmitted as data. That is to say, rather than setting and monitoring signal levels as is done with a standard RS-232 interface, RFCOMM uses commands and responses to communicate the state of the multiplexed serial interface (thus virtualizing the RS-232 signals).

RS-232 defines other states that are not directly represented by signals. Notable among these is the baud rate, or the clock frequency used to transmit and receive data. In standard cabled serial communications, a clock governs the time associated with the signal transition to and from low and high levels, which define the 0/1 bit patterns. Obviously both sides of the interface must use the same clock frequency, or baud rate, to correctly interpret the data that is transmitted across the wire. For wireless environments, however, there is no cable and thus no signal wire to pulse at a specified frequency. Clearly, though, Bluetooth wireless communication does employ clock timings to communicate over the air-interface at the baseband level. Since RFCOMM operates over the transport layers of the protocol stack, it makes use of the packet structure and transmission medium used by those lower layers. The

baud rate of Bluetooth wireless communication is determined by the packet types and structures being sent over the air-interface. The actual communication will occur at the rate determined by the baseband protocol, regardless of what baud rate might be specified at the RFCOMM layer for serial port emulation. So while an application or other client of RFCOMM can specify a baud rate (this would be a typical action, especially for legacy applications, and RFCOMM allows it), the specified baud rate does not determine the actual data rate. In many cases, the data transmission rate using Bluetooth wireless communication could be faster than for typical cabled communications.

RFCOMM Protocol Usage

Curiously, the RFCOMM chapter of the core specification (volume 1) includes a section containing the sort of information (application considerations, interactions with other protocol stack layers and SDP service record data) that is usually found in the profile specifications (volume 2). This is an artifact of the development of the RFCOMM specification. As previously noted, RFCOMM specification development was underway almost from the beginning of the SIG's formation. Along with the lower-layer transport protocols, which consumed much of the SIG's attention at first, RFCOMM was one of the first protocols to reach a stable specification level (this is due partly to the fact that RFCOMM leverages the TS 07.10 standard and partly to the hard work applied to the RFCOMM specification by its owners in the SIG, since the SIG recognized that RFCOMM was a key element of the version 1.0 protocol stack and a foundation upon which other protocol layers and profiles were to be built). Most of the profiles were developed after the core specification was stable. The forward-looking authors of the RFCOMM portion of the specification had already included some of the information that subsequently became part of the serial port profile (covered in Chapter 14).

So the RFCOMM chapter of the specification gives some hints on using this layer of the protocol stack. The specification talks about *Port Emulation* and *Port Proxy* entities, the former mapping platform APIs to RFCOMM functions and the latter mapping RFCOMM to a "real" RS-232 external interface. The point, though, is that the authors of the RFCOMM specification not only have specified a protocol that is necessary for many legacy applications to make use of Bluetooth wireless communications but also have offered a few considerations for the applications that use that protocol.

IPR2020-00202
Apple Inc. EX1057 Page 185

In fact the programming model suggested in RFCOMM is a specific instance of the generalized model suggested in the section entitled "The Application Group" in Chapter 5 (refer back to Figure 5.4). In this case we suggest a thin layer of Bluetooth adaptation software for legacy applications that maps platform APIs to specific functions of the Bluetooth protocol stack. In the case of RFCOMM, which provides an emulated serial port, this adaptation software (which the specification calls a port emulation entity) needs to map the application's interactions with a "real" RS-232 serial port to the equivalent operations for the RFCOMM emulated serial port. For the most part these are expected to be initialization operations such as activating and configuring the serial port and establishing a serial connection; and finalization operations such as terminating the serial connection. Once a general serial port adaptation layer is in place in a system, all those legacy applications that use serial communication ought to be enabled to use Bluetooth transports via the RFCOMM emulated serial port.

As pointed out earlier, the use of serial ports is prevalent in devices and environments where Bluetooth wireless communication is likely to be used, and the majority of the version 1.0 profiles depend upon serial port communications. In the absence of a version 1.0 specification for general networking, the RFCOMM protocol provides an important utility for legacy applications. The implementation of this protocol, along with adaptation software for legacy applications that use serial communications, permits many simple cable replacement applications of Bluetooth wireless communication.

The Service Discovery Protocol (SDP)

Service discovery is a process by which devices and services in networks can locate, gather information about and ultimately make use of other services in the network. In traditional networks such as LANs, these services might be statically configured and managed by a network administrator. In these environments, the administrator or end user performs the configuration that is necessary for one participant in the network to use the services of some other network member. For example, a PC user might specify all of the information associated with a network e-mail service (including the mail server name, user and account names, e-mail type, capabilities and options, and so on) to the PC's operating system and applications; once all this information is entered into the PC

IPR2020-00202
Apple Inc. EX1057 Page 186

and associated with that e-mail service, then the e-mail service becomes available to the PC user.

Administered network services of this sort may be fine for many fixed networks but are really not suitable for temporary mobile networks (ad hoc networks) such as those that might be formed using Bluetooth wireless communication. In these environments a more dynamic, flexible and adaptive solution is needed. Graham, Miller and Rusnak [Graham99] observe the growing incidence of these ad hoc networks and the resulting demand for self-configuring systems:

> The emergence of information appliances and new types of connectivity is spurring a new form of networking: unmanaged, dynamic networks of consumer devices that spontaneously and unpredictably join and leave the network. Consumers will expect these ad hoc, peer-to-peer networks to automatically form within the home, in very small businesses and in networked vehicles. …
>
> To achieve the goals of simplicity, versatility and pleasurability, the appliances and the network(s) they join must *just work* right *out of the box*. By *just work* we mean that the participants on the network must simply *self configure*. By *self configure* we mean that these network devices and services simply discover each other, negotiate what they need to do and which devices need to collaborate **without any manual intervention**.[5]

Protocols for service discovery can help to enable this self-configuration. Since much of the interdevice communication in Bluetooth usage scenarios is of a peer-to-peer, ad hoc nature, the SIG determined that a service discovery protocol in the stack could provide significant value. The resulting protocol, known as SDP, is a central component of nearly all of the profiles and usage cases, both existing and foreseen.

The service discovery concept is not new or unique to Bluetooth wireless technology. Numerous service discovery technologies are available in the industry, some of them well known. As is evident in other layers of the protocol stack, the SIG is content to adopt existing protocols where it makes sense to do so. In the case of service discovery, though, the SIG developed its own protocol unique to and optimized for Bluetooth wireless communication rather than adopting some other service discovery protocol in the industry. The reasons should become evident as we review SDP's development in the next section.

5. Reprinted by permission from *Discovering Devices and Services in Home Networking*, copyright (1999) by International Business Machines Corporation.

SDP Development

The need for a service discovery component in the protocol stack was recognized early in the process of developing the specification, although direct work on SDP did not commence until later. In early and mid 1998, many of the initial participants in the SIG were focusing on the transport protocols and key middleware protocols like RFCOMM. While the need for other protocols had been identified, task forces of experts to develop these protocols had not been assembled in all cases. In the case of SDP, some preliminary work had been started at Intel and Ericsson in the summer of 1998.

In early internal versions of the specification, service discovery was a section within the L2CAP part of the specification. Initially, L2CAP channels were modeled after a computer bus, and service discovery was concerned exclusively with the transport of Plug and Play parameters over this virtual bus. In September 1998, at a SIG meeting in London,[6] author Bisdikian suggested that the addition of a transport protocol for Plug and Play parameters unnecessarily complicated the L2CAP specification, and that such a protocol merited its own service discovery portion of the Bluetooth specification.

In October 1998 the SIG held a developers conference in Atlanta which author Miller attended. Based upon conversations during that conference, Miller was asked to chair the service discovery task force of the SIG's software working group shortly thereafter. The following month the newly constituted group met for the first time as a formal SDP task force.

While at this time (late 1998) many of the protocol layers had been under development for several months, with some of them approaching levels of stability that would soon near final stages, SDP was still really in a nascent state of forming the requirements and the beginning of a proposal to address those requirements.

Among the identified objectives for Bluetooth SDP were:

Simplicity: Because service discovery is a part of nearly every Bluetooth usage case, it is desirable that the service discovery process be as simple as possible to execute. For the SDP task force this also implied the reuse of other Bluetooth protocols to the extent possible.

6. To advance the development of the specification, face-to-face meetings among SIG members have taken place in many different countries reflective of the multinational constituency of the SIG membership.

IPR2020-00202
Apple Inc. EX1057 Page 188

Compactness: As described in previous chapters, the formation of Bluetooth communication links between two devices can in some cases be time consuming. Since service discovery is a typical operation to perform soon after links are established, the SDP air-interface traffic should be as minimal as feasible so that service discovery does not unnecessarily prolong the communication initialization process.

Versatility: The version 1.0 specification includes a number of profiles, and future revisions will undoubtedly add to the list, which is expected to continue to grow. Since an exhaustive set of profiles, usage cases and associated services cannot be foreseen or accurately predicted, it is important for SDP to be easily extensible and versatile enough to accommodate the many new services that will be deployed in Bluetooth environments over time. To support this objective the SDP task force chose a very broad definition of "service," so that the widest possible spectrum of features (services) could be supported in the future.

Service location by class and by attribute: In the dynamic ad hoc networking environment it is important to enable client devices and users to quickly locate a specific service when they already know exactly (or at least largely) what they are looking for. It should be straightforward to search for a general class of service (say, "printer"), for specific attributes associated with that service (for example, "color duplex IBM printer") and even for a specific instance of a service (such as a specific physical printer).

Service browsing: In addition to searching for services by class or attribute, it is often useful simply to browse the available services to determine if there are any of interest. This is a different usage scenario than is searching for specific services, and in some respects it is almost a contrary objective, but the SIG agreed that both usage models have merit, and they developed a solution that uses a consistent method to support both specific service searching and general service browsing.

These objectives led to the development of requirements for a simple, flexible protocol and data representation for service discovery in the protocol stack. Popular industry discovery protocols were reviewed, but none seemed to provide a good match for the SDP objectives–many of these technologies provide robust and comprehensive service discovery and access methodologies, but the SIG was really looking for a fairly low-level, simple, narrow-in-scope solution that met the rather modest objectives noted above in a straightforward manner.[7] At this

point Motorola® approached the SIG with a proposal to contribute some technology, suitable for use in Bluetooth service discovery, that Motorola had had under development for several years. Through a contributing adopter agreement Motorola was then able to participate in the SDP task force of the SIG (and in fact the Motorola representative served as editor of the SDP specification), with their contributed technology forming the basis for SDP.

So with the SDP effort underway in November 1998, the SDP requirements and scope were agreed upon and the specification development ensued, incorporating the ideas contributed by Motorola along with the many contributions by the other SIG member companies. Even though the real SDP work started later than for many other protocols, through hard work the SDP specification was completed, ratified and published along with the bulk of the other protocols in the stack in July 1999 in the version 1.0 specification.

The following sections describe some of the key facets of the SDP specification, including why these elements are significant and the rationale for including them in the specification.

SDP Examined

Key to understanding the development of SDP is to understand its motivation and requirements. In fact, this information is included at the beginning of the SDP portion of the specification. As noted above, SDP is intended to allow devices in Bluetooth environments to locate available services. As the specification states, these environments are qualitatively different from traditional networks such as LANs or WANs. Devices and services are likely to come and go frequently in Bluetooth piconets. Thus SDP was developed to satisfy the requirements of such environments.

Some of the notable requirements for SDP are listed in the preceding section. These are also mentioned and expanded upon in the specification. Also of interest are those items that SDP does not attempt to address, at least in version 1.0 of the specification.[8] The "Non-Requirements and Deferred Requirements" portion of the SDP specification

7. Subsequent to publication of the version 1.0 specification, efforts were begun to map some of the leading industry service discovery technologies to the Bluetooth stack. Chapter 16 gives details of this work.

8. Of these, the Bluetooth SIG might choose to enhance SDP in the future to address some of the issues. Many, however, are likely to remain outside the scope of Bluetooth SDP, since some of the issues can be and are addressed by industry discovery protocols, which Bluetooth SDP can accommodate, as explained in the main body text.

IPR2020-00202
Apple Inc. EX1057 Page 190

can be summarized largely with the statement that SDP is narrow in scope, focusing primarily on discovery in Bluetooth environments and leaving more sophisticated service functions and operations to other protocols which might be used in conjunction with SDP.

SDP includes the notion of a client (the entity looking for services) and a server (the entity providing services). Any device might assume either role at a given time, acting sometimes as a service client and sometimes as a service provider (server).

The service provider needs to maintain a list of *service records* that describe the service(s) it provides; this list is called the *service registry.* A service record is simply a description of a given service in a standard fashion as prescribed by the specification. A service record consists of a collection of service attributes containing information about the class of the service (which might be printing, faxing, audio services, information services, and so on), information about the protocol stack layers that are needed to interact with the service, and other associated information such as human-readable descriptive information about the service. Figure 8.3 illustrates the general structure of a service registry with its constituent service records. Shown is a set of services, each with a service record handle (depicted by srvRecHnd[0] through srvRecHnd[j]) and a set of attributes per service (shown as srvAttribute[0:a] through srvAttribute[j:c]). Further explanation of the content of these service records follows.

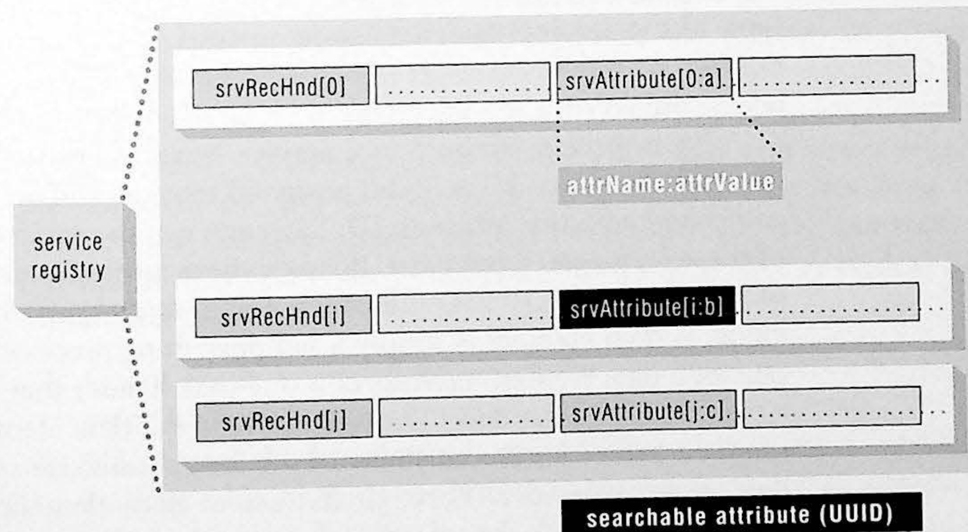

Figure 8.3
General SDP service registry structure.

IPR2020-00202
Apple Inc. EX1057 Page 191

Service records consist of both *universal service attributes* and *service-specific attributes.* The universal service attributes are simply those parts of the service record that apply to all types of services, such as the service class and protocol stack information noted above. Service-specific attributes are those parts of the service record that are relevant only for a specific class or instance of a service. Examples of service-specific attributes could include attributes specific to a printing service (such as color, duplex and finishing capabilities), attributes specific to an audio service (such as data rate or encoding scheme) or attributes specific to a dial-up networking service (such as serial port configuration or modem setup information). Volume 1 of the specification includes definitions for a set of universal service attributes (those which could apply to all types of services), but it does not include service-specific attributes, since it would be impossible to specify and predict all of the attributes for every imaginable type of service. Service-specific attributes are defined in profiles (volume 2 of the specification). Since profiles describe a usage scenario and how the protocol stack is used, they effectively define a service. So, for example, the headset profile defines the service specific attribute "remote audio volume control" that applies to the headset service. While the universal service attributes can apply to all types of services, this does not mean that they are mandatory–it is not required that every service include every universal service attribute in its service record. In fact, only two of the universal service attributes are mandatory: the service class attribute, which defines the class, or type of the service, and the service record handle, which serves as a pointer, or reference to the service record and is used by the client to access the server's service record.

Each service attribute in a service record consists of an attribute identifier (attribute ID, a 16-bit unsigned integer) and an attribute value associated with that attribute ID. Each entry of the service record is one of these (*attribute, value*) pairs. Because these attributes describe all sorts of information, SDP uses the concept of a *data element* for the attribute value. A data element is simply a self-describing piece of data. The first part of a data element consists of a one-byte header that tells the actual type and size of the data. The remainder of the data element consists of the data values for the attribute, of the format and size specified by the data element header. Through the use of data elements, SDP allows attribute values to be of several types, including strings, Booleans, signed and unsigned integers of various sizes, and universally unique identifiers (UUIDs, discussed further below). Moreover, these data types can be lists of the scalar elements noted above, thus providing a

flexible representation for the many data types of which attribute values might be composed.

Discovering a service in Bluetooth wireless communication reduces to a simple operation: the client specifies the service(s) of interest and the server responds, indicating any available services that match what the client specified. In practice for SDP this consists of the client's sending a request in the form of an SDP protocol data unit (SDP_PDU) that indicates what service(s) it is searching for and the server's sending back a response, also in the form of an SDP_PDU, that indicates what services match the request that the client has made. To accomplish this, the client needs a standard way to represent the service(s) of interest and the server needs a standard method to match its available services against the client's specification. For this purpose SDP introduces *universally unique identifiers* (UUIDs).

A UUID is a concept adopted from the International Organization for Standardization (ISO). UUIDs are 128-bit values that can be created algorithmically and, generally speaking, can be virtually guaranteed to be entirely unique–no other UUID ever created anywhere will have the same value.[9] One advantage of using UUIDs is that new identifiers can be created for new services without requiring a central registry of identifiers maintained by the SIG, although the SIG does include a list of "well-known" UUIDs in the specification for those services related to the published profiles. So a client looking for a service just specifies the UUID associated with that class of service (or with the specific service) in its service search request, and the service provider matches that UUID against those of the services it has available to generate its response.

The SDP_PDUs exchanged between the client and server are simple transactions. The general SDP protocol flow requires only two transactions; the specification defines three different SDP transactions, but the third is really just a composite of the first two. A typical SDP transaction consists of:

1. Client sends a request to search for service(s) of interest; server responds with handles to services that match the request.
2. Client uses the handle(s) obtained in step 1 to form a request to retrieve additional service attributes for the service(s) of interest.

9. This concept is sometimes hard to grasp, but universally unique identifiers can in fact be created. While there is an extremely small chance of duplication, UUIDs as defined by ISO (see [ISO96]) are quite sufficient for the purposes of Bluetooth SDP and turn out to be quite valuable in this context.

IPR2020-00202
Apple Inc. EX1057 Page 193

Following the above transaction, the client will presumably use the information obtained in step 2 to open a connection to the service using some protocol other than SDP to access and utilize the service. Step 1 is called the *ServiceSearch* transaction and consists of the *ServiceSearchRequest* SDP_PDU from the client to the server and the *ServiceSearchResponse* SDP_PDU in return (from server to client). As noted above, the ServiceSearchResponse SDP_PDU contains handles to one or more services that match the request. In step 2, the client presents one or more of those handles in a *ServiceAttributeRequest* SDP_PDU which causes the server to generate a *ServiceAttributeResponse* SDP_PDU; this exchange is the *ServiceAttribute* transaction. In the ServiceAttributeResponse SDP_PDU will be the attribute values associated with the service that correspond to the attribute IDs that the client specified in the ServiceAttributeRequest SDP_PDU. These attributes may be a combination of universal service attributes and service-specific attributes, and in most cases should provide the client with enough information to subsequently connect to the service.

The specification defines a third SDP transaction, called the *ServiceSearchAttribute* transaction. This transaction consists of a *ServiceSearchAttributeRequest* SDP_PDU from the client to the server followed by a *ServiceSearchAttributeResponse* SDP_PDU from the server to the client. It is actually redundant to the first two transactions described above and is included for efficiency. What the ServiceSearchAttribute transaction allows is the combination of steps 1 and 2. That is, the client can form a single request that specifies not only services to search for but also the attributes to return for matching services in the server's response. The server then responds with handles to matching services as well as the requested attribute values for those matching services. An implementer thus has a choice between the two alternatives for SDP transactions.[10] More importantly, though, the ServiceSearchAttribute transaction may in some cases be more efficient in terms of the number of bytes transmitted over the air-interface. The consolidated transaction itself requires more bytes than the individual transactions but could result in fewer total transactions. Especially in cases where many service records are being accessed, such as in a service browsing application, the ServiceSearchAttribute transaction might be more efficient.

10. It should be noted, however, that different profiles mandate the use of different SDP transactions, so if a profile is being implemented, the profile will determine which SDP transaction(s) need to be used, and the programming effort to support all three transactions should not be great.

IPR2020-00202
Apple Inc. EX1057 Page 194

In a nutshell, this is most of what is needed for SDP transactions. The specification also includes protocol definitions for special cases, including an error response SDP_PDU and a mechanism, called the *continuation state*, for dealing with server responses that cannot fit into a single SDP_PDU.[11] The syntax of these protocol transactions and the data elements that they carry is detailed in the specification and is not reproduced here. A unique feature of the SDP chapter of the specification is the inclusion of several detailed protocol examples as an appendix to the SDP specification. The members of the service discovery task force of the SIG who developed the specification felt that because the actual byte streams generated for SDP transactions can be complex (even though the transactions themselves are conceptually simple), it would be useful to include the examples as a guide for implementers. The complexity is introduced mostly when complex data elements (such as *DataElementSequences*, which are lists of data elements and which can be nested) are carried in the SDP_PDUs. When these complex data types are included in SDP_PDUs, or when SDP_PDUs need to be split using the continuation state information, the various "count" fields that introduce segments of the SDP_PDUs need to accurately reflect the number of bytes that follow in that segment. The examples in the specification serve to clarify the correct construction of SDP_PDUs.[12]

Figure 8.4 summarizes the SDP transactions. Shown in the figure are representations of the relevant arguments and parameters passed in the SDP_PDUs, although these are not complete lists of all arguments and parameters; the complete syntax is in the specification. As Figure 8.3 shows, only services and service attributes that are described by UUIDs are searchable. Attributes of a service which are not described by UUIDs are not searchable and can be retrieved only after a service has been located using a UUID attribute.

---

11. The client can specify the maximum size for the response to its request SDP_PDU. It is possible for the response that is generated by the server to be larger than this maximum size. In this case, the server includes some continuation state information at the end of its response, which allows the client to initiate another request to obtain the next portion of the response, if desired.
12. In fact the developers of the specification learned first hand of the need for these examples when they constructed them, since there were some errors in the first internal versions of the examples. There were even some errors in the examples published in the original version 1.0A SDP specification, which were subsequently corrected in version 1.0B.

IPR2020-00202
Apple Inc. EX1057 Page 195

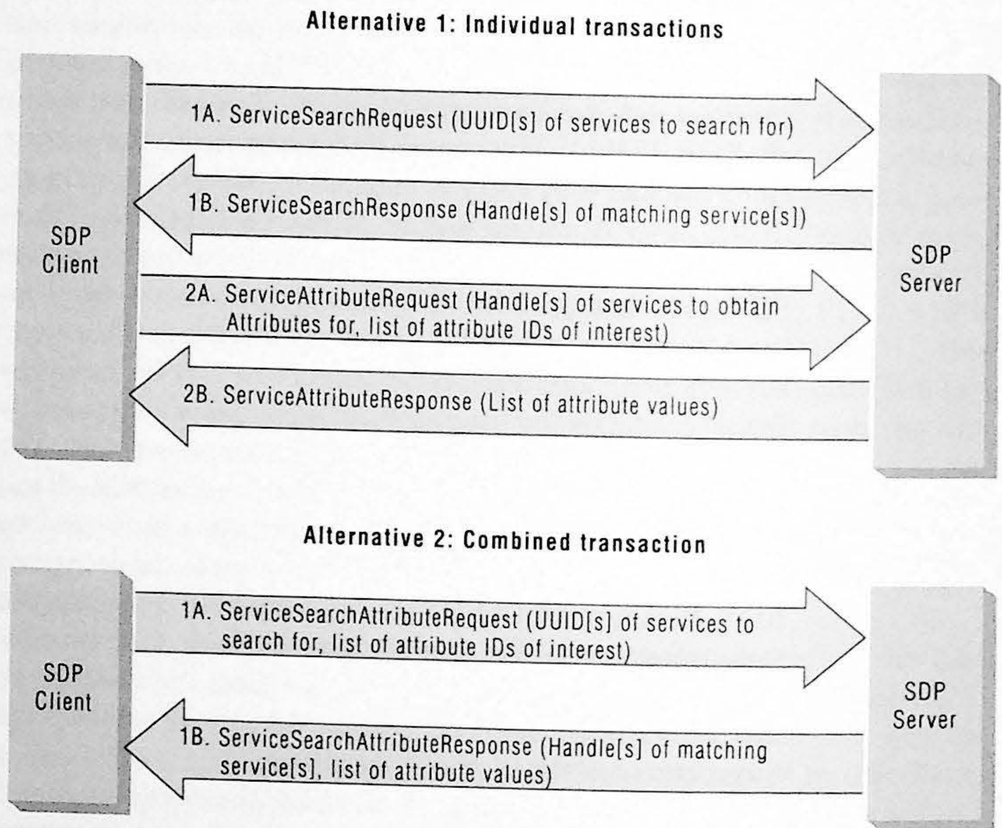

Figure 8.4
SDP transaction summary.

SDP Usage

Since SDP was developed primarily for discovering services in Bluetooth environments, the applications most likely to make use of SDP will be those developed specifically to be aware of Bluetooth wireless communication (as opposed to legacy applications). One exemplary application for SDP usage is what we will call the *Bluetooth Piconet Minder*[13] (or BPM application). Such an application is likely to be included, in one form or another, in many Bluetooth devices. A BPM application as we envision it would present a view of available devices and services in proximity (in a piconet) to the user and to other applications. This could include a user interface; one might imagine icons or other representations of devices and services. Such an application could

13. This term is used generically here and is not known to, or intended to, conflict with any actual product names.

IPR2020-00202
Apple Inc. EX1057 Page 196

give a user a central point to manage the Bluetooth connections to other devices and to select and make use of the services offered by those other devices. To support such functions, a BPM application might make use of service searching and service browsing and thus initiate SDP transactions to populate the service information that is exposed to the user and to other applications.

Certainly other applications designed for use in Bluetooth environments might use SDP. Every profile (or at least every "non-generic" profile that involves concrete usage scenarios) includes an SDP service record to be used when implementing that profile. Applications written specifically to exercise the protocol stack will probably need to execute SDP transactions to successfully instantiate the profiles. First, such applications will need to execute SDP transactions with another device to determine if that device offers the desired service, and if so, those applications will need to execute additional SDP transactions to obtain the information from the service record about how to access that service (this can include such information as the required protocol stack and associated parameters that the service uses).

In the case where multiple applications use SDP (perhaps one or more profile applications and a BPM application), it may be advantageous to implement a central SDP client and SDP server that are available to all the applications that need them. These SDP "helper applications" could be implemented as part of the common services layer that was described in Chapter 5. The applications could use the platform APIs to access the common SDP services which would generate the SDP transactions and pass the retrieved information back to the applications.

The Service Discovery Application Profile (SDAP) detailed in Chapter 12 offers guidance for application interactions with SDP. While, as previously noted, the specification does not define APIs, the SDAP does define primitive operations that could be mapped to APIs and events on many platforms, thus providing a basis for SDP common services.

There may be legacy applications that make use of service discovery, but such applications probably use some other industry discovery protocol (perhaps Jini™, Universal Plug and Play™, Salutation™, Service Location Protocol, the IrDA service discovery protocol, or some other protocol). Since SDP was developed for Bluetooth applications, legacy applications would not be expected to include this protocol without modification to the application. Even for these applications, though, SDP does offer some accommodation. One of the design points for SDP

IPR2020-00202
Apple Inc. EX1057 Page 197

was to ensure that other popular industry discovery protocols could be used in conjunction with it. One of the things that can be discovered using SDP is that the service supports one or more other discovery protocols. Thus SDP might be used in the initial service discovery phase to locate the service; further SDP transactions might be used to discover that the service supports, say, Salutation; once this has been determined, the newly discovered protocol (Salutation in the example) can be used for further interaction with the service. SDP specifically supports this sort of operation. A SIG white paper [Miller99] describes how Salutation can be mapped to SDP. Similar mappings to other technologies should be possible, and the SIG is working toward formalizing some of these mappings as profiles, as discussed in Chapter 16.

IPR2020-00202
Apple Inc. EX1057 Page 198

9

IrDA Interoperability Middleware Protocols

Continuing our examination of the middleware protocols, we now visit those protocols intended to provide interoperability with IrDA applications. In this chapter we examine the protocols and conventions that together in the specification are termed "IrDA Interoperability." This term does not mean that devices with Bluetooth wireless communication can communicate directly with IrDA devices but rather it refers to protocols that enable common applications to use either form of wireless communication. The IrDA interoperability middleware includes protocols adopted from the Infrared Data Association (IrDA), namely IrOBEX (or briefly, just OBEX), its associated data object formats and the Infrared Mobile Communications (IrMC) method of synchronization. IrDA interoperability occupies fewer than 20 pages in version 1.0 of the core specification (volume 1), although over 100 pages are devoted to the IrDA-related profiles in volume 2 (this latter material is discussed in Chapter 14 of this book). As is the case with RFCOMM, the main reason that the IrDA interoperability protocols take up only a relatively few pages of the specification is that they call out existing standard protocols that are fully described in external documentation, in this case, specifications of the IrDA. IrDA application interoperability is a fundamental design principle for Bluetooth wireless communication, and thus the IrDA interoperability middleware is a key element of the protocol stack. These protocol layers are the basis for several profiles which are likely to become some of the most commonly implemented and widely deployed scenarios on devices that employ the Bluetooth technology. This chapter examines not only the IrDA

IPR2020-00202
Apple Inc. EX1057 Page 199

interoperability information in the specification (as usual, attempting to reveal the rationale for these protocols) but also the similarities and differences between IrDA and Bluetooth wireless communication, since this latter topic is fundamental to the design basis for the IrDA interoperability solution adopted for the specification.

The IrDA interoperability protocols reside above other middleware protocols. In particular OBEX operates over RFCOMM in the standard case and also could operate over TCP/IP as an optional implementation.[1] Like RFCOMM and SDP, the OBEX session protocol uses a protocol data unit (PDU) structure, allowing the higher layers of the stack to work with logical data elements at a higher level of abstraction than that of the packet formats used by the transport protocols or even by RFCOMM. More importantly, though, OBEX primarily is intended to promote application interoperability with IrDA, so applications using this protocol with IrDA wireless communication can adapt easily to the use of Bluetooth links. Figure 9.1 depicts OBEX in the protocol stack. As shown in the figure, OBEX is used by higher layers of the stack, typically applications, and OBEX in turn uses other middleware protocols, namely RFCOMM (and optionally, within the constraints described above, also may use TCP/IP).

1. While TCP/IP operation for IrOBEX is described in the Bluetooth specification (volume 1) in the same level of detail as is RFCOMM operation for IrOBEX, there is no SIG-specified end-to-end TCP/IP solution for version 1.0, since the specification does not address the general use of TCP/IP over Bluetooth transport protocols. Thus, even though IrOBEX could work over TCP/IP, and the IrDA Interoperability chapter of the specification describes how to do this, IrOBEX is assumed to operate over RFCOMM throughout volume 2 of the specification because TCP/IP is not fully specified in the version 1.0 Bluetooth protocol stack. As noted elsewhere, though, TCP/IP can operate over PPP links, and general IP networking solutions are in progress within the SIG.

IPR2020-00202
Apple Inc. EX1057 Page 200

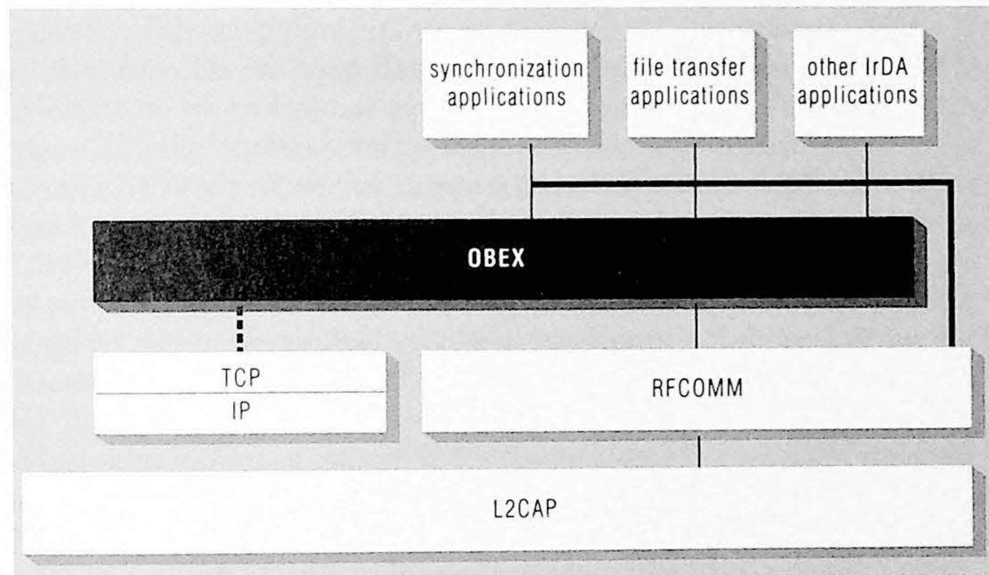

Figure 9.1
OBEX in the Bluetooth protocol stack.

IrDA and Bluetooth Wireless Communication Compared

Before examining the specification in detail, we first look at the underlying technologies of IrDA and Bluetooth wireless communication. The objective, after all, for including IrDA protocols in the stack is to promote interoperability with IrDA applications (hence the "IrDA Interoperability" in the title of this chapter and in the specification). Here we must clearly state that this interoperability is at the application layer, not at the physical layer. IrDA interoperability does not imply that Bluetooth devices can communicate directly with IrDA devices. Instead it is intended to promote development of applications that can use either transport. Previous chapters have touched on the relationship of IrDA and Bluetooth wireless communication. Here we compare and contrast specific features of these technologies. Some excellent background reading on this topic can be found in [Suvak99], which was written by a SIG and IrDA participant.

IrDA is a specific use of infrared light as a communications medium; Bluetooth technology is a specific use of radio waves as a communications medium. Like the Bluetooth SIG, the Infrared Data Organization (IrDA) specifies hardware and software protocols for wireless communication intended to promote interoperable applications.

IPR2020-00202
Apple Inc. EX1057 Page 201

While both technologies are wireless, they use different parts of the electromagnetic spectrum with quite different signal propagation characteristics. Since infrared uses the nonvisible infrared light spectrum, IrDA communication is blocked by obstacles that block light (such as walls, doors, briefcases and people). The signal wavelength used with Bluetooth communication, at about 12.5 cm, is three orders of magnitude greater than that of IrDA. At this wavelength, 2.4 GHz RF communications can penetrate these sorts of obstacles. Recent advances in infrared technology have enabled more diffuse transmission patterns, although much of the IrDA equipment in use today uses a relatively narrowly focused beam, which usually requires that the two devices engaged in IrDA communication be aligned with (pointed at) each other. RF transmission patterns are generally spherical around the radio antenna, so any two devices within range can communicate with each other whether or not they are "pointed at" each other (in fact, the second device might not be visible at all to the user of the first device, as it could be in another room behind doors and walls or even on another floor of a building, for example).

The IrDA specification is more mature than the Bluetooth specification, having been available for several more years. IrDA technology was already widely deployed when the Bluetooth specification was first released. Thus IrDA has already undergone several phases of enhancement that Bluetooth wireless technology might undergo in the future. Among these is some improvement in communication speed. The initial IrDA data rate of 115 Kbps has now been enhanced to 1 Mbps, which is comparable to that of the first Bluetooth radios. Today IrDA can achieve data rates of up to 4 Mbps, with even higher rates already specified. While Bluetooth RF communication has the potential for similar increased data rates (and the SIG is investigating these possibilities; see Chapter 16 for more information), it is likely to lag the IrDA speeds for at least several years.

The effective range for Bluetooth wireless communication is about 10 meters using the standard 0 dBm radio. With optional power amplification of up to 20 dBm, range on the order of 100 meters can be achieved. IrDA range is about 1 meter and, as noted above, generally requires a line of sight to establish a connection.

Bluetooth wireless technology is designed for very low power consumption, and as compared to other RF technologies it consumes very little power. IrDA communication, however, consumes even significantly less power than Bluetooth technology, since far less power is required for infrared transceivers than for RF transceivers.

IPR2020-00202
Apple Inc. EX1057 Page 202

In terms of cost, IrDA hardware was significantly less expensive than Bluetooth radio modules at the time Bluetooth technology was introduced. This is partly due to the maturity and wide deployment of IrDA: the technology has been around long enough that several iterations of cost optimizations have occurred, and installed IrDA units number in the millions, so economies of scale resulting from high-volume production have been realized. Bluetooth modules in the year 2000 had not realized either of these forms of cost reduction; even so, it is expected that Bluetooth hardware is likely to remain more expensive than IrDA hardware[2] owing to the complexity of the underlying technology, although the cost difference probably will narrow over time.

Table 9.2 summarizes these feature comparisons of IrDA and Bluetooth wireless communication. The table reflects the state of these technologies in the year 2000 and reflects consensus forecasts in the industry. The table is intended to be illustrative rather than authoritative; certain parameters will vary from implementation to implementation.

Table 9.1
Illustrative feature comparison of IrDA and Bluetooth wireless communication. Estimates as of 2000; some values are implementation dependent.

Feature	IrDA technology	Bluetooth technology
Connection establishment	Line of sight	Penetrates obstacles
Transmission pattern	Relatively narrow conical	Relatively spherical
Data rate	4 Mbps	1 Mbps
Range	1 meter	10–100 meters
Power consumption	10 mW (nominal)	100 mW (estimated nominal; product-dependent)
Transceiver module cost	< $1.00	$5.00 (estimated, approximately 2003–2005)

As explained in the specification and in the following sections, IrDA and Bluetooth wireless communication share similar application

2. Bluetooth radio module costs in the 2003–2005 time frame are variously estimated to approach $5 U.S.; IrDA solutions are already well below this figure.

IPR2020-00202
Apple Inc. EX1057 Page 203

domains, even though the underlying technology used to achieve scenarios such as object exchange and synchronization is inherently different. Feature differences may cause one technology to be preferred over the other in certain environments and applications, although we believe that both have merit and both are likely to be deployed in pervasive computing devices in the foreseeable future. Thus the IrDA interoperability provisions of the specification can help to enable the best use of either or both technologies as the situation warrants.

The IrDA Interoperability Protocols

One of the most common applications for IrDA technology is exchanging files and other objects. This includes exchanging electronic business cards between two devices as well as transferring files and other data objects between two devices, all in an ad hoc fashion and without wires. The devices commonly used in these scenarios–especially mobile phones, notebook computers and handheld computers, but also others[3]–are the same set of devices where Bluetooth technology is deployed or is expected to be deployed. Even though direct communication between an IrDA device and a Bluetooth device is not feasible, it seems clear that these same sorts of applications are quite relevant and useful in Bluetooth environments. In fact, profiles exist in the version 1.0 specification for both object push (which could be used for electronic business card exchange) and file transfer (which can include transferring several specific object types). An extended application of this sort of data exchange is synchronization, where the data is not only exchanged but is also replicated between the two devices. A profile exists for this usage case also, and it too is based upon the IrDA application and protocols. In volume 2 of the specification, all of the file transfer, object push and synchronization profiles are derivatives of the generic object exchange profile, and all of these are described in further detail in Chapter 14 of this book.

The rationale for IrDA interoperability in Bluetooth wireless communication is to enable the same applications to operate over both IrDA and Bluetooth links, and the most straightforward way to do this is to use the same session protocol in both environments. Since the IrDA protocols already existed and some were suitable for Bluetooth applications, the SIG chose to adopt OBEX and IrMC in the Bluetooth

3. Perhaps including digital cameras and computer peripherals such as printers.

IPR2020-00202
Apple Inc. EX1057 Page 204

protocol stack in the same relative position as in the IrDA protocol stack. Unlike RFCOMM, the IrDA interoperability specification does not include any significant subsets, alterations, adaptations or clarifications of OBEX, although there are some specific considerations (such as calling out the specific OBEX version 1.2) noted for its use in the protocol stack. Much of the description in the specification echoes important elements of OBEX and describes precisely how OBEX is used over other middleware layers of the protocol stack.

IrDA Interoperability Protocol Development

The reuse of IrDA protocols and specifically OBEX was identified as the design direction of the SIG early in the specification's development. As with RFCOMM, work was underway on the use of OBEX and IrDA protocols and data formats at about the time the SIG was publicly announced. The synchronization usage case was already identified at that time, as were file transfer and data exchange applications (the latter scenarios at that time were part of the conference table usage case). In early 1999 the business card exchange scenario had led to the beginning of what is now the object push profile; file transfer and synchronization were well defined, and work on profiles for these usage models was also underway. It quickly became evident that a generic framework profile that applied to all IrDA interoperability usage cases (that is, all those profiles using OBEX) would be valuable, so the generic object exchange profile also was initiated.

Given the objective of interoperability between IrDA and Bluetooth applications, an initial goal of the SIG was to produce a specification that would allow a single application to operate seamlessly over both wireless transports. The SIG was (and is) motivated to reuse existing protocols where appropriate. These considerations led to the selection of OBEX as the point in the IrDA protocol stack that could be inserted into the corresponding point in the Bluetooth protocol stack to allow applications to deal with the same protocol (OBEX) in both environments. With study and discussion in the SIG it was determined that OBEX could operate both over RFCOMM, which was reasonably well defined by this time, and over TCP/IP (although the latter is enabled only in certain circumstances in version 1.0, as discussed below). Other transports for OBEX not directly applicable to the Bluetooth protocol stack include IrSock (infrared sockets), IRCOMM and Tiny TP (or TTP), some of which are mentioned in passing in the specification.

IPR2020-00202
Apple Inc. EX1057 Page 205

The OBEX Protocol Examined

IrDA interoperability in general, and OBEX and IrMC in particular, are significant elements of the protocol stack, yet relatively few pages[4] are devoted to the topic in volume 1 of the version 1.0 specification. OBEX is the basis for several of the version 1.0 profiles and IrDA interoperability is an important objective and key value of the Bluetooth technology. The IrDA interoperability specification can be so compact because of the SIG's decision to adopt existing IrDA protocols that are fully specified by IrDA (the IrDA OBEX specification [IrDA99a] is about 85 pages long while the full IrMC specification [IrDA99b] is nearly 200 pages, although only a portion of this latter specification deals directly with IrMC synchronization).

While the specification discusses the use of OBEX over TCP/IP, it does not define how TCP/IP should operate natively over Bluetooth transports. The fact that OBEX can operate over TCP/IP will become more important in the future when the SIG defines general TCP/IP operation over Bluetooth links (as described in Chapter 16). Until such a definition exists, the fact that OBEX can operate over TCP/IP transports is not directly relevant for version 1.0 implementations.[5] Thus TCP/IP operation for OBEX is specified as optional.

The other Bluetooth protocol over which OBEX is designed to operate is RFCOMM. RFCOMM (detailed in the previous chapter) was designed specifically with OBEX in mind as one of the RFCOMM clients. Since OBEX over TCP/IP is defined only in the context of PPP for version 1.0, we focus here on its use over RFCOMM. The specification describes the requirements for the use of OBEX over RFCOMM. These are not new or unique requirements specific to Bluetooth environments; rather they define the boundaries within which a generic OBEX application should operate to ensure that it will work over RFCOMM and thus over Bluetooth transports. Among the considerations for OBEX over RFCOMM are:

---

4. At fewer than 20 pages, the IrDA Interoperability portion of the specification is easy to read from beginning to end, yet it is a fairly complete description of how IrDA protocols are used in the Bluetooth stack.
5. Which is not to say that such a stack could not be implemented; in fact, it could. But like all other implementations not based upon profiles, the risk of noninteroperability exists. Because TCP/IP is such an important protocol, it is safe to assume that TCP/IP over Bluetooth links eventually will be solved (this is being pursued by the SIG), and thus it is good to know that OBEX over TCP/IP is already enabled.

IPR2020-00202
Apple Inc. EX1057 Page 206

Client and server functions: The specification indicates that both client and server functions must be supported by devices implementing the OBEX IrDA interoperability protocol. When one examines the IrDA interoperability profiles (see Chapter 14), it becomes evident that while it is technically possible for a device to support only a client or only a server role, it is really useful only when a device can support both roles. Even object push, which is largely a one-way data transfer, still requires both a client role (in this case the client needs to push the object) and a server role (in this case the server needs to pull the object). This apparent dichotomy is explained in Chapter 14.

RFCOMM multiplexing: All OBEX transactions must use a separate RFCOMM server channel (as described in the previous chapter, only one RFCOMM connection is permitted between two devices); thus, multiple clients of RFCOMM must use its protocol multiplexing feature. The OBEX server must open a separate RFCOMM channel connection with a client. Similarly the RFCOMM connection needs to be terminated when the OBEX session that uses it is terminated. The specification also describes how to parse the stream-oriented communications that occur over RFCOMM to delimit the OBEX packet structures contained therein.

SDP Support: OBEX applications in Bluetooth environments need to be able to make use of SDP. OBEX clients need to obtain the relevant information about the OBEX service from its service record in the OBEX server. OBEX servers need to populate the service record with information such as the appropriate RFCOMM server channel to use. As described in the previous chapter, this SDP application enablement might be obtained through the use of common SDP application services; these need not be unique to OBEX applications.

OBEX provides a session protocol for transactions between two devices. The IrDA defines both connection-oriented and connectionless sessions; the Bluetooth specification calls for use of only the connection-oriented sessions, since this is what best fits Bluetooth environments. Like SDP, OBEX transactions consist of a request PDU issued by the client followed by a response PDU issued by the server. With OBEX, the client role normally is assumed by the device that initiates the transaction, while the responding device becomes the server. Also similar to SDP, the OBEX PDUs consist of a header, a size indicator and the

arguments and parameters associated with the particular transaction. Fundamentally, OBEX is a simple protocol, with the main operations being *connect* and *disconnect* to initialize and terminate sessions, along with *get* and *put* operations to exchange data objects within an existing session. These operations are described in the Bluetooth specification and detailed in the IrOBEX specification.

In addition to a session protocol, OBEX also serves as an object transport for the data that can be exchanged in OBEX sessions. To support the IrDA interoperability profiles, the specification calls out particular object formats as follows:

vCard: format managed by the Internet Mail Consortium [IMC96a] for representing electronic business cards.

vCalendar: format managed by the Internet Mail Consortium [IMC96b] for representing electronic calendar and schedule entries.

vMessage: format defined by IrMC [IrDA99b] that represents electronic messages and electronic mail.

vNote: format defined by IrMC [IrDA99b] that represents short electronic notes.

Volume 2 of the specification calls out each specific object format as it applies to the object push, file transfer and synchronization profiles. Some of these profiles allow for different versions of the object types noted above; some also allow for other generic object types to be used with OBEX.

IrMC Synchronization Examined

In addition to transferring data objects over OBEX, it is also quite useful to *synchronize* these same objects. Synchronization, generally, is the process of comparing two sets of data and then updating those data sets so that they exactly reflect (are synchronized with) each other at the point in time that the synchronization is performed. There are variations on the synchronization process, such as one-way synchronization where a "slave" data set is always updated to match a "master" data set, or partial synchronization where only a subset of the data is synchronized, but in general the idea is to merge the changes made in two (or even more) data sets into each other so that the data sets become replicas of each other (until additional changes are made to them). Synchronization allows data (perhaps calendar entries, address books or e-mail)

IPR2020-00202
Apple Inc. EX1057 Page 208

to be manipulated at various times and places and then be replicated against some other related data set so that the updates from the data manipulation can be applied. Applications for synchronization include synchronizing address books to incorporate new, changed or deleted entries; synchronizing calendar entries to incorporate new and changed schedule items; and replicating e-mail to send and receive new notes and messages and incorporate saved or deleted messages. Synchronization can be especially useful when these types of data are kept on more than one device. Address books, calendars and e-mail can be replicated among mobile phones, handheld computers, notebook computers and network repositories of data so that no matter which device is used, the data on that device can be current and updates to these data can be reflected on the other devices through synchronization.

Note that the devices mentioned above are some of the devices most likely to employ Bluetooth wireless communication. Thus it seems that synchronization is a natural usage case in Bluetooth environments. Note further that the types of data mentioned above as being common candidates for synchronization (calendar, e-mail and contact information) are the same data types defined in the profiles for object transfer over OBEX. Thus it seems evident that it ought to be valuable and feasible to employ OBEX-based synchronization in the specification, and indeed this is precisely what the SIG has done. Just as with object transfer, the SIG has chosen to adopt the method defined by the IrDA, called IrMC synchronization. IrMC synchronization builds upon the OBEX session protocol and certain object formats (including some object formats defined by IrMC itself) to specify a method of synchronizing these objects. As with OBEX, the specification incorporates IrMC synchronization as a way to enable IrDA application interoperability.

The core specification (volume 1) includes very little information about synchronization per se, focusing instead on the use of OBEX in Bluetooth wireless communication. It does, however, briefly describe Bluetooth synchronization when discussing the synchronization profile, which is where the details can be found. Essentially IrMC provides a framework for OBEX-based exchange of data; given this capability to exchange data formats including those noted above, additional logic can be applied to perform differencing and selective object transfer, thus accomplishing synchronization using the IrMC framework within OBEX sessions. Chapter 14 more fully explores Bluetooth synchronization.

IPR2020-00202
Apple Inc. EX1057 Page 209

IrDA Interoperability Usage

The IrDA interoperability information in the core specification (volume 1) includes a description of the related profiles found in volume 2 of the specification. In fact, since IrDA interoperability is really about application interoperability, there is a larger amount of information (over 100 pages) on this topic in the profiles than in the core specification. Recall that IrDA interoperability just makes reference to existing IrDA specifications and describes how to use these standards in Bluetooth environments.

The reuse of IrDA protocols, along with the fact that these protocols operate over RFCOMM (a serial port abstraction), is intended to facilitate the use of existing IrDA applications in Bluetooth environments. IrDA applications are familiar with the use of serial port communications and are likely to have support for OBEX protocols. By accommodating these IrDA interoperability layers in the Bluetooth stack, the SIG has paved the way for applications that can operate with both IrDA and Bluetooth wireless communication.

IPR2020-00202
Apple Inc. EX1057 Page 210

10

Audio and Telephony Control

Support for voice or, more generically, audio is a distinguishing attribute of Bluetooth wireless communication. With support for both voice and data, the technology is well positioned to bridge the domains of computing and communications, as evidenced by the enthusiastic support for the Bluetooth technology within both industries. Several of the profiles address scenarios in which both a computing device and a telephony device are used. This chapter, our final in-depth examination of the core specification, deals with the components of the protocol stack that enable telephony and voice (audio) communication. The telephony control protocol is embodied by the *TCS-BIN* (or just TCS for short) layer, while audio can be carried natively over the baseband. TCS is based upon the existing ITU-T Q.931 protocol [ITU98], but even so it occupies over 60 pages in the specification. TCS is a binary encoding for packet-based telephony control and resides above the L2CAP layer of the stack. TCS-BIN is sufficient to realize the version 1.0 telephony profiles, although applications using AT commands over the RFCOMM serial port abstraction (including headset, dial-up networking and fax) might also accomplish a form of telephony control (this latter form of telephony control is not included as a separate entity in the version 1.0 specification; it is discussed further in subsequent sections here). Audio is not a layer of the protocol stack per se but rather a specific packet format that can be transmitted directly over the baseband layer. Since audio is frequently (although not exclusively) associated with telephony applications, it is discussed together with TCS in this chapter as a logical convenience. This chapter examines telephony

IPR2020-00202
Apple Inc. EX1057 Page 211

functions, including audio, in Bluetooth wireless communication. As in preceding chapters we will not only provide highlights and interpretations of the specification but also touch upon the background information for these elements of the protocol stack, including the evolution of TCS-BIN.

Figure 10.1 depicts audio and TCS-BIN in the protocol stack; it also shows the component we call *AT Command Telephony Control.* This latter component is a remnant of what was once called *TCS-AT* and is explored further below. In general, when we refer simply to TCS we mean the TCS-BIN layer of the stack. TCS-BIN resides above L2CAP; audio communicates directly through the baseband; and AT command telephony control operates over RFCOMM. Telephony control applications can communicate directly with TCS-BIN and might also use AT command telephony control.

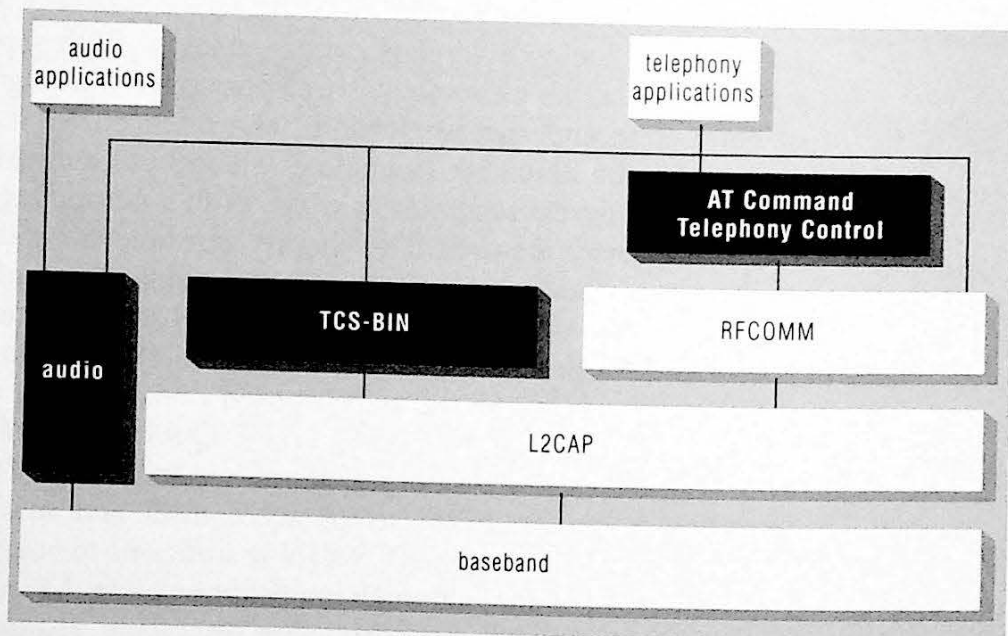

Figure 10.1
Audio and TCS-BIN in the Bluetooth protocol stack. Also shown is the AT command based form of telephony control used by some applications.

Audio and Telephony Control Operation

TCS-BIN is used for the call control aspects of telephony, including establishing and terminating calls along with many other control functions that apply to telephone calls. TCS can be used to control both

IPR2020-00202
Apple Inc. EX1057 Page 212

voice and data calls. When a voice call is made the audio element of the stack is used to carry the its content; in the case of data calls the data content can be carried over the transport layers of the stack (perhaps also involving other middleware layers). The call control functions provided by TCS-BIN can be used no matter what the call content (voice or data) is; data calls like those used with the dial-up networking profile are supported and so is voice telephony, like that used for the cordless telephony and intercom profiles.

TCS-BIN also defines a method for devices to exchange call signaling information without actually having a call connection established between them; this is called *connectionless TCS* and is described more fully below. Another aspect of TCS-BIN is that of ***group management*** functions. When there is a group of devices that all support the TCS-BIN protocol, the members of the group (called a ***wireless user group***, or WUG) can make use of some special functions defined by TCS, including group membership management, telephony service "sharing" among devices in the group and a method for a fast direct connection between two group members. The TCS-BIN call control and other functions are examined more fully below.

A second form of call control, which we have called AT command telephony control, was introduced above. While it is not defined as a named protocol in the specification, it is mentioned here because it is a well-known method for accomplishing call control, and it is used by several profiles. In fact, at one time this concept was embodied as a separate protocol and element of the stack called TCS-AT. While TCS-AT is no longer defined as a separate entity (and indeed, given the existence of TCS-BIN, a separate SIG-defined TCS-AT protocol is unnecessary, as described more fully below), it is worth acknowledging that this sort of telephony control does exist in many Bluetooth environments. AT commands are modem control commands that are likely to be used especially by legacy applications; these applications typically are configured to communicate with a modem over a serial port. Within the Bluetooth protocol stack these applications could use RFCOMM to communicate with a compatible modem service using the same AT command call control functions as in other environments, with little or no change to the application (especially through the use of a Bluetooth adaptation layer as described in Chapter 5). TCS-BIN is the only telephony control protocol defined as a separate entity in the specification, and it is the protocol upon which several telephony profiles are based. However, AT command-based telephony control is also used in the

IPR2020-00202
Apple Inc. EX1057 Page 213

headset, fax and dial-up networking profiles, even though no separate AT protocol is specified by the SIG.

Audio, as already pointed out, is not really a layer of the protocol stack. In fact it would not be unreasonable to consider audio as a specialized sort of transport layer, since it is largely embodied as a particular packet format that is sent and received directly over the air-interface using the baseband protocol. Indeed, outside the baseband chapter, the specification directly addresses audio only in an appendix that is fewer than ten pages long! Yet we have established that voice support is a key differentiating value of Bluetooth wireless communications, and clearly audio directly supports voice (voice and audio are often equated, although voice is not the only form of audio). So why does the specification not contain a chapter on audio with a description and page count commensurate with the importance of audio for Bluetooth applications? The answer has already been suggested: because Bluetooth audio is really just a specification of a packet format and an encoding scheme for the data in those packets, it does not require a lengthy explanation. Once the allowances (including time slot reservation and audio packet definition, described more fully in Chapter 6) have been made at the baseband layer to support audio traffic, little more specification is required. In fact the actual bulk of the audio specification can be found in the baseband chapter of the specification, which even includes a section devoted entirely to audio baseband traffic. Thus to fully understand Bluetooth audio one should understand the baseband protocol stack layer, described in Chapter 6 of this book. However, because audio so often is associated closely with voice and thus with telephony, it is logically consistent to discuss it here along with the other telephony-related functions.

TCS Protocol Development

Telephony control is intertwined with audio functions, and in fact it was audio that drove the need for telephony control rather than the other way around. Before there was a TCS working group, it was agreed that the protocol stack needed to support audio so that voice as well as data traffic could be enabled. At first the audio requirement pointed out the need for some control functions, which initially were presented as "audio control" functions. These audio control capabilities were needed to support the ultimate headset, speaking laptop and three-in-one phone usage models (described in Chapter 3), and initially just a small set of simple operations (such as make a call, answer a call, terminate a

IPR2020-00202
Apple Inc. EX1057 Page 214