11

Communicating via Infrared: Using Sockets

Chapter 10 introduced sockets as a way to access upper layer protocols through a consistent, modular programming interface. Programming the IR port is a great place to start using sockets. To program the IR port at a level different from IRCOMM, we need to access IrDA protocols implemented by Symbian OS. This is done using sockets, and this chapter will demonstrate the concepts we developed in Chapter 10 via this kind of IR port communication.

We will start by reviewing the IrDA protocols that are available for use and their properties. We will then overview definitions Symbian OS has constructed for these protocols. Then we will look at socket programming through the IR port: we will see the stages we saw in Chapter 10 from two sides—the sender side, which will look like a client, and the receiver side, which will look like a server.

11.1 A Quick Protocol Review

Back in Chapter 4, we discussed the protocols that have been developed for using the IR port. Along with the IRCOMM protocols, there are three protocols that are useful for user applications:

- *IrMUX:* This protocol implements a datagram-based communication channel between devices. IrMUX connections reflect the properties of datagram-based connections: unreliable data communication that does not rely on acknowledgement. Multiple IrMUX connections can coexist over a single IR port.
- *TinyTP:* This protocol implements a reliable connected channel between devices. It uses error control to detect problems and

maintains a connection between endpoints until data exchange is completed. Multiple TinyTP connections can coexist over the same IR port.

- *IrTranP:* This protocol implements a TinyTP connection for the exchange of digital images, for example, with digital cameras.

In addition to these protocols, we mentioned the IAS—the Information Access Service—that acts as a service database for the IR-based device.

Before we embark on this tour of IrDA sockets, I think it is appropriate to ask the question "why?". Why complicate communications when IRCOMM is relatively simple, has relatively low overhead, and mimics other, well-understood protocols? The answer is that these upper level protocols provide functionality that IRCOMM does not: error control, flow control, and the multiplexing of IR services by multiple applications through the same port. By using datagrams to implement data transfer, these protocols can be faster, more error free, and can share the IR port in ways IRCOMM was not designed to implement.

11.2 System Definitions for IrDA

11.2.1 Addressing Definitions

To use IrDA sockets, we need a set of system definitions. These definitions clarify the general system concept of a socket and focus specification on IrDA.

The sockets we will be using will be created using addresses from the `KIrdaAddrFamily` group and have the properties specified by the `TIrdaSockAddr` class. IrDA addresses are integers, but there are other characteristics that are associated with IrDA devices that identify them. These are grouped in the `SIrdaAddr` structure:

```
struct SIrdaAddr
{
  TUint iHostDevAddr;
  TUint iRemoteDevAddr;
  TBool iSniff;
  TBool iSolicited;
  TUint8 iIrlapVersion;
  TUint8 iFirstServiceHintByte;
  TUint8 iSecondServiceHintByte;
  TUint8 iCharacterSet;
  TUint8 iServiceHintByteCount;
  TUint8 iHomePort;
```

```
  TUint8 iRemotePort;
  TUint8 iSpare;
};
```

From this structure, we can see the two integer device addresses: local and remote. The `iSniff` variable tells whether the device is capable of "sniffing" or discovering other devices. The `iSolicited` variable tells whether the device at this address initiated the connection or was solicited by another for the connection. Of the other variables, the service hint bytes in `iFirstServiceHintByte` and `iSecondServiceHintByte` are of interest. These bytes depict the level of IrLMP support we can expect from the device at this address. Together, these bytes form 16 bits whose value specifies what services are available. Finally, there are two integers denoting the ports on which the services are identified. These allow multiplexing of data streams over the same connection by tying an identifying integer (the port number) onto each data stream.

Service Hints

The service hint bytes contain IrLMP support information. There are twelve different services supported in version 6.1 of Symbian OS. These are detailed in the file `\Epoc32\include\ir_sock.h`. Examine this file, and look for the service definitions in the section labeled "EXPORTED CONSTANTS". Service support definitions in this file are masks (such as `KIrPnPMask`) that are ORed together (for example, `KIrPnPMask | KIrFaxMask | KIrModemMask`) to represent the set of services provided by a device's IR support.

11.2.2 IrDA Socket Options

There are several options that can be read or set for IrDA sockets. Of immediate interest are two of these:

```
const TUint KUnexpeditedDataOpt        = 0;
const TUint KExpeditedDataOpt          = 1;
```

These values determine the urgency of data written to a socket. Unexpedited sending is the default; expedited sending must be specifically set on the socket if it is to be used. Expedited sending of data implies the data is important, and the data packets are given priority over others that go out over the IR connection (see section 11.11).

Complete Definitions

Among the many constant definitions in `\Epoc32\include\ir_sock.h` is a list of options that you can set on IrDA sockets. Look for a group of these definitions whose names end in "`Opt`". `KUnexpeditedDataOpt` and `KExpeditedDataOpt` are defined in this file.

11.3 Initializing the Socket Server Connection

The first step in connecting two devices over IR is system initialization. For IrDA, this means connecting to the socket server.

Connecting to the socket server is a simple operation. We need to have an instance of the `RSocketServ` class, and we need to call its `Connect()` function. We also need to nudge it to load the protocol we will be using.

The `TodoX` example uses TinyTP to transport data. From the `TodoX` code, socket server connection looks like

```
// Load the physical and logical device drivers
result = User::LoadPhysicalDevice(PDD_NAME);
if (result != KErrNone && result != KErrAlreadyExists)
   User::Leave(result);
result = User::LoadLogicalDevice(LDD_NAME);
if (result != KErrNone && result != KErrAlreadyExists)
   User::Leave(result);

// Start comm server if necessary
result = StartC32();
if (result != KErrNone && result != KErrAlreadyExists)
   User::Leave(result);

// Connect to the socket server
result = socksvr.Connect();
User::LeaveIfError(result);

// Load the protocol: TinyTP
result = socksvr.NumProtocols(nProtocols);
User::LeaveIfError(result);
result = socksvr.FindProtocol(_L("IrTinyTP"),protocolInfo);
User::LeaveIfError(result);
```

First, note that we still had to load the physical and logical device drivers. Even though we are not directly interfacing with them, they will be used by the IrDA protocol stack. Second, note the way we load the `IrTinyTP` protocol module. We tell the socket server to

find the protocol, which forces it to load the right module if it is not present. This function call also fills in the information about the protocol we were looking for in the variable `protocolInfo`, which we will use later in socket calls. We could have looked for "`IrMUX`", which would have given us a socket that used IrMUX.

11.4 Opening the Socket

Opening the socket is straightforward—a call to `OpenL()` in the `RSocket` class. We must specify what kind of protocol we want, but this was derived for us when we looked up the protocol module and loaded it.

In the `TodoX` example, we define a new `OpenL()` function as follows:

```
TBool TodoXferIRSocket::OpenL()
{
   TInt result;

   result = sock.Open(socksvr, protocolInfo.iAddrFamily,
      protocolInfo.iSockType, protocolInfo.iProtocol);
   User::LeaveIfError(result);

   return true;
}
```

We use information stored in `protocolInfo` that we derived previously to give us information about address family, socket type, and protocol. These parameters are all derived from the fact that we chose the TinyTP protocol as the method for socket communication.

11.5 Configuring the Socket: Device Discovery

Configuring a socket is different than configuring a device. With serial communications, we had a real hardware device with which to work. With socket programming, we have a software socket abstraction and layers of protocol with which to work.

With IrDA sockets, configuration means *device discovery*. We need to force the device we are working with to look for other IR devices that can be sending or receiving the data we expect. We did not need to do this for serial communication, because there was only one device on the other side, and we did not care what its "address" was. Symbian OS implements device discovery

through the name resolution interface, using an instance of the `RHostResolver` class and its `GetName()` function.

In the `TodoX` example, device discovery is implemented in the `Configure()` function:

```
void TodoXferIRSocket::Configure()
{
   TInt result;
   THostName hostname;

   TPckgBuf<TUint> buf(1);
   sock.SetOpt(KDiscoverySlotsOpt,KLevelIrlap,buf);

   result = resolver.GetByName(hostname,iHostEnt);
   resolver.Close();
   User::LeaveIfError(result);

   addr = TIrdaSockAddr(iHostEnt().iAddr);
}
```

Notice that we had to set up the socket to do device discovery using the IrLAP protocol through the `SetOpt()` function call. Using a one-element array, we set up discovery to have a single slot, i.e., we will be looking for one device. Further, note that the `GetByName()` function I chose to use is the synchronous form. Using the asynchronous form would mean that this should be part of the active object implementation (in `RunL()`).

After the call to `Configure()` completes successfully, device discovery has been performed and we have a compatible device that might be willing to connect. Note that at the end of the `Configure()` function, the address of the remote device is derived from the `TNameEntry` object using `iHostEnt()`. We can derive a bit more information from this address, as seen from the functions below taken from the `TIrdaSockAddr` class:

```
TUint GetRemoteDevAddr() const;
TUint GetHostDevAddr() const;
TBool GetSniffStatus() const;
TBool GetSolicitedStatus() const;
TUint8 GetIrlapVersion() const;
TUint8 GetCharacterSet() const;
TUint8 GetFirstServiceHintByte() const;
TUint8 GetSecondServiceHintByte() const;
TUint8 GetServiceHintByteCount() const;
TUint8 GetHomePort() const;
TUint8 GetRemotePort() const;
```

These are self-explanatory and correspond to the `SIrdaAddr` structure we discussed earlier. Since that structure is private in the

TIrdaSockAddr class, these functions are the only way to get at this information.

A data receiver will not do device discovery. It is not the discoverer; it must be discovered. In Chapter 10, we discussed this type of situation. The data receiver is a server and must configure itself to listen for a connection from a data sender. The server will configure the socket by binding the socket locally, setting up a listening queue on the socket, and opening a second socket to connect to the incoming connection. In the TodoX example, this looks like

```
void TodoXferIRSocket::ConfigureReceiver()
{
   TInt result;
   THostName hostname;

   // Create the listener socket
   iListener.Open(socksvr, protocolInfo.iAddrFamily,
      protocolInfo.iSockType, protocolInfo.iProtocol);
   addr.SetPort(KIRPortNumber);
   iListener.Bind(addr);
   iListener.Listen(5);

   // Create a blank socket for later
   iServiceSocket.Open(socksvr, protocolInfo.iAddrFamily,
      protocolInfo.iSockType, protocolInfo.iProtocol);
}
```

11.6 Exchanging Data

Now we have created a socket and either discovered a device that is waiting for a connection or fielded a connection request and responded. We must engage in the protocol phases we have outlined in previous chapters: initiate the protocol, exchange data, and terminate the protocol.

We should use active objects to exploit the asynchronous nature of the I/O functions we will be using. For these active objects, the fact that we are using sockets will add some states to the state diagrams that we constructed for serial communication. For sockets, we added some new steps to the connection process: we added an asynchronous call to a Connect() function for senders/clients and a call to Accept() for receivers/servers. This changes our state diagram; we need an initial connect or accept state. For the TodoX example, Figure 11.1 shows the new state diagram for senders, adding a new SEC state.

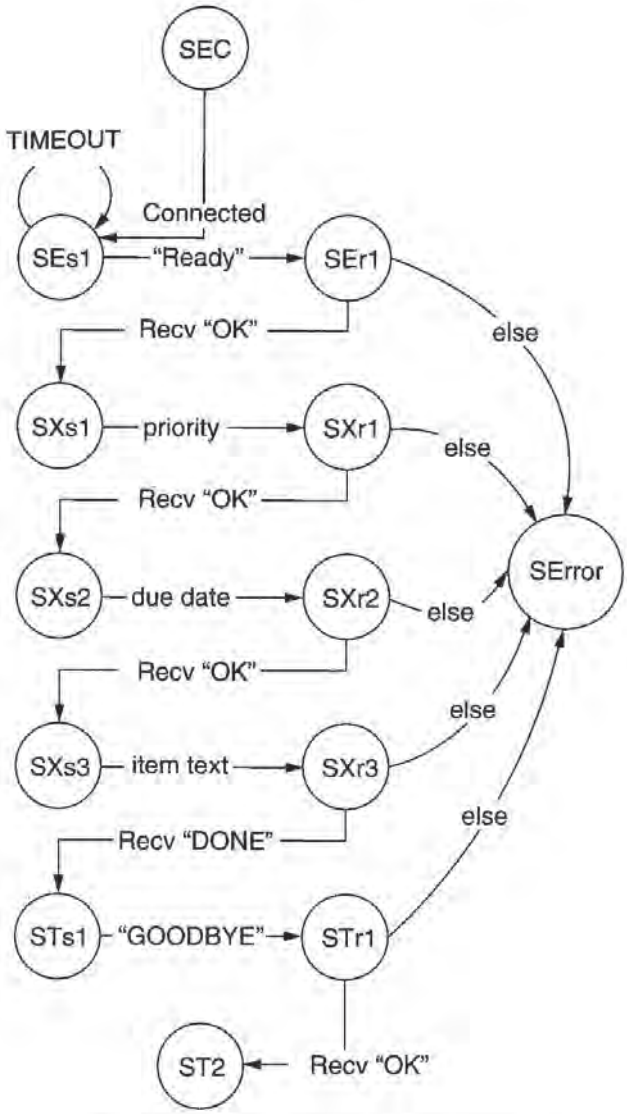

Figure 11.1 New `TodoX` state diagram.

11.6.1 Initiating the Protocol

For socket-based communication, initiating the data exchange protocol means connecting the socket to the remote device in addition to sending initial protocol synchronization data. For the `TodoX` example, we must connect a channel, then send our "Ready" string. We start by connecting to a port on the remote device. Recall that a port number is used to identify the data stream that is being serviced by a specific remote process. We identify which port we are binding to, assuming the receiver is binding to the same port on its side.

The `SendItem()` function from the `TodoXferIRSocket` class in the `TodoX` example illustrates the sender sequence nicely:

```
void TodoXferIRSocket::SendItem(CAgnEntry *aEntry)
{
   iEntry = aEntry;

   CParaFormatLayer *iParaFormatLayer =
      CParaFormatLayer::NewL();
   CCharFormatLayer *iCharFormatLayer =
      CCharFormatLayer::NewL();

   // Get to-do item information
   iTodoItem = CAgnTodo::NewL(iParaFormatLayer,
      iCharFormatLayer);
   iTodoItem = (CAgnTodo *)(aEntry->CastToTodo());
   priority = iTodoItem->Priority();
   duedate = iTodoItem->DueDate();

   // Set up the address and connect
   addr.SetPort(KIRPortNumber);
   sock.Connect(addr,status);

   iSendingState = SEC;
   SetActive();
}
```

This looks a lot like the code from the active objects we used to implement serial communication. The difference is that now we engage an initial connection step prior to diving into the data exchange.

On the receiver side, we must start with a call to `Accept()` and wait for a sender. The `ReceiveItem()` function in the `TodoXferIRSocket` class from the `TodoX` example implements this:

```
void TodoXferIRSocket::ReceiveItem()
{
   CParaFormatLayer *iParaFormatLayer =
      CParaFormatLayer::NewL();
   CCharFormatLayer *iCharFormatLayer =
      CCharFormatLayer::NewL();

   iTodoItem = CAgnTodo::NewL(iParaFormatLayer,
      iCharFormatLayer);
   iListener.Accept(sock,status);

   iReceivingState = REA;
   SetActive();
}
```

Once we have connected and completed the `Accept()` call, the active object's `RunL()` function is called, and we can enter into the

data exchange as we did with the serial active object implementation. For the TodoX sender, this fragment of code implements this:

```
switch (iSendingState) {
  case SEC:
    if (status == KErrNone) {
      iAppUi->SetProgress(_L("Establishing protocol"));
      buffer.Copy(_L8("Ready"));
      Send(buffer);
      iSendingState = ESEs1;
    } else {
      User::Leave(status.Int());
    }
    break;
```

This should look familiar from the way the serial versions started the data exchange. The receiver side is analogous, moving the protocol initialization code to the RunL() function.

11.6.2 Exchanging Data

Once we have connected the sockets end-to-end and started the protocol, the interface and data sequence for socket I/O is very much the same as for serial I/O. While you can use Read() and Write() functions in the same way as with serial solutions, using Send() and Recv() functions are recommended because they are more flexible.

As an example, consider the Send() function from the TodoX example:

```
void TodoXferIRSocket::Send(const TDes8 &aText)
{
   TInt len;
   TBuf8<100> buffer;

   len = aText.Length();

   // Send the length of the buffer
   buffer.SetLength(0);
   buffer.Append((TChar)(48+len));
   buffer.Append(aText);
   sock.Send(buffer, len+1, status);

   // Get active!
   SetActive();
}
```

This code is very close to the serial version but uses Send() rather than Write(). There may be occasions when we need to use

the flags in the `Send()` or `Recv()` functions. These flags differ depending on the context in which they are used.

11.6.3 Shutting Down the Protocol

For IrDA sockets, shutting down the protocol requires no special socket operations. While initiating the protocol required connecting the socket, there is no special "disconnect" operation. Shutting down the socket automatically disconnects it from the remote endpoint.

11.7 Terminating the Socket Connection

As we discussed in Chapter 10, the socket might need to be "drained" of requests before we close it. Also, the connection to the socket server needs to be shut down.

The `Close()` function from the `TodoX` example illustrates this:

```
void TodoXferIRSocket::Close()
{
   sock.CancelAll();
   socksvr.Close();
   sock.Close();
}
```

We could have used the asynchronous version of `Close()`, i.e., `Shutdown()`. This would have done the same thing, but we would have had to weave it into the `RunL()` function definition to respond to its completion.

11.8 Using IAS

The Information Access Service (IAS) provides a database that holds information about IrDA connections and their properties. You can query this database, even on the remote device, to find out more information about the IrDA connection. Queries can be made after device discovery has been successfully completed.

IAS queries use two classes we have not yet mentioned. The first is `RNetDatabase`, a class that is used to generically access socket connection information, and `TIASQuery`, a class that specifically constructs IAS database queries.

Queries are made up of two components: an IAS class and an IAS attribute. The IAS class is not a C++ class; it is a configuration

specifier. (I would like to rename this unfortunate definition, but it is a standard.) Classes have attributes, and there can be multiple attributes per class. A table of some of these is given below:

Table 11.1 IAS Classes

IAS Class Name	IAS Attribute Name	Response
Device	DeviceName	Remote device hostname
	IrLMPSupport	Level of support by remote device for IrMUX
IrDA:IrCOMM	Parameters	Collection of parameters detailing IrCOMM support
	IrDA:IrLMP:LsapSel	Range of remote port numbers for IrCOMM service
	IrDA:TinyTP:LsapSel	Range of remote port numbers for TinyTP service
	IrDA:InstanceName	Name of IAS instances on remote device
IrDA:IrOBEX	IrDA:TinyTP:LsapSel	Range of remote port numbers for IrOBEX service

To make an IAS query, we need to construct `RNetDatabase` and `TIASQuery` objects, build the query using the `TIASQuery` object, and call the `Query()` function of the `RNetDatabase` object. The query will include both the IAS class and attribute names. Consider the code below:

```
TIASResponse TodoXferIRSocket::IAS(TDesC8& aClassName,
   TDesC8& aAttributeName)
{
   TInt result;
   RNetDatabase rnd;
   TIASQuery query;
   TIASResponse response;

   result = rnd.Open(socksvr,  protocolInfo.iAddrFamily,
      protocolInfo.iProtocol);
   User::LeaveIfError(result);

   query.Set(aClassName, aAttributeName,
      addr.GetRemoteDevAddr());
   rnd.Query(query, response, status);
   User::WaitForRequest(status);

   return response;
}
```

Notice the call to `Query()` is provided a status variable to monitor and a `TIASResponse` response variable to fill with the query

response. By now you can figure out that `Query()` is an asynchronous function.

The status variable will give `KErrNone` for successful completion of a query. However, two other values are significant here: `KErrBadName` will be returned if there is no remote class with the name given, and `KErrUnknown` will be returned if the class name is present but the attribute is not there.

The return value stored in the `TIASResponse` object is one of four types.

- Type 3 results are strings.
- Type 2 results are sequences of bytes.
- Type 1 results are integers.
- Type 0 results are "missing" results—they are not given because of errors.

IAS queries are meant to provide information from the remote device as to how it is providing service and how it will support protocols. This information is useful especially in heterogeneous environments where the level of support from the (unknown) remote device cannot be assumed.

11.9 Using IrMUX

In the `TodoX` example from the previous sections, we used TinyTP as the protocol of choice for IrDA communications. We chose this because it is easier to use—it is more reliable and fewer errors result from transmission. However, IrMUX is also a protocol available to you when programming IrDA communications.

IrMUX is a datagram-based protocol. It implements an unreliable, connectionless data transfer between two sockets. While sockets using IrMUX are programmed using the same general functions we have discussed, there are a few differences.

- *Loading protocol modules:* The IrMUX protocol module is loaded with the name "IrMUX" to the `RSocketServ::FindProtocol()` function.
- *Sending data through the socket:* As we stated in Chapter 10, the most appropriate way to send data through connectionless sockets is to use the `SendTo()` and `RecvFrom()` functions. These functions can be used with the address of the discovered device as an argument.

- *Socket shutdown:* While an IrMUX socket is terminated the same way a TinyTP socket is, the implementation is different. While Symbian OS will cause a connected TinyTP socket to finish any remaining data requests, it will simply cut off and discard any data I/O pending for connectionless IrMUX sockets.

IrMUX—and connectionless sockets in general—are typically used in situations where the higher overhead per packet is not necessary. This includes situations where there is a lot of data traffic or the data is comprised of single, simple datagram messages.

Using `Connect()` with Connectionless Sockets

If you read the system documentation for Symbian OS version 6.1, you will find references to using connected socket methods with connectionless sockets. The documentation implies, for instance, that `Read()` and `Write()` may be used with IrMUX sockets.

Do you believe this implication? To check this out for yourself, you need to answer some questions:

- If you use `Read()` and `Write()` with IrMUX sockets, how do you specify the device address that will receive the datagrams?
- One answer to the previous question is to use `Connect()` as you do with connected sockets to "preregister" the device address. Can you use `Connect()` with connectionless sockets?

The way to verify your answers to the above questions is to (a) inspect the `ir_sock.h` file and documentation for clues and (b) code up some experiments and see for yourself. What would you expect to see if `Read()` and `Write()` would *not* work with connectionless sockets? What would you expect to see if they did?

11.10 Using IrTranP

We have covered both TinyTP and IrMUX protocols. A third protocol that we have defined for IrDA is IrTranP, a protocol for the exchange of digital pictures. This protocol is not implemented via a socket interface. Rather, the implementation uses a class object—the `CTranpSession` class—and implements the entire protocol from within this object. This class uses a special

MTranpNotification class, which contains a set of callback functions that are called at various stages in the picture exchange process. This MTranpNotification class is an abstract class; you must provide a class derived from it, implementing the callback functions, to the CTranpSession class object when you instantiate it. Once the CTranpSession class is set up, a call to its Get() function will get the exchange session rolling. As the exchange progresses, the following callback sequence is used:

- When a successful connection is made to a remote device, the Connect() callback function is called in the MTranpNotification object.
- After connection, the remote device will start sending the picture data, and the CTranpSession class object will make repeated calls to the ProgressIndication() callback function of the MTranpNotification object to relay progress information.
- When the data exchange is complete, the CTranpSession class object will call the MTranpNotification::GetComplete() callback function.
- When the connection to the remote device is broken, a call to the Disconnected() callback function of the MTranpNotification object is made.

11.11 A Word About Expedited Socket Output

At the beginning of this chapter, we mentioned that IR output through sockets could be done in one of two modes: unexpedited or expedited. The former mode is the default; to use expedited sending of data, an option must be specifically set on the socket. To do this, we have two choices. The first would use the SetOpt() function from the RSocket class and specify the KExpeditedDataOpt option value, as follows:

```
sock.SetOpt(KExpeditedDataOpt,&buffer,KLevelIrMUXSAP);
```

This, however, is not the recommended way. The recommended way gives more control to the programmer to make both expedited and unexpedited sends over the same socket. To use this technique, we would use the Send() function from the RSocket class, using the flags in the call to specify expedited I/O:

```
Sock.Send(buffer, KsockWriteUrgent, status);
```

11.12 Summary

In this chapter, we have discussed the methods we need to use to implement non-serial communication across the IR port. We accessed these upper layer IrDA protocols through the use of sockets, thereby providing a great illustration of the concepts developed in Chapter 10. We examined each stage in socket usage and applied each stage to IrDA communications, using both TinyTP and IrMUX protocols. We concluded by briefly reviewing IrTranP and expedited IR output.

In the next chapter, we move from point-to-point communication into the world of multipoint communication. We examine TCP/IP protocols and how we use them in Symbian OS.

12

Communicating with TCP/IP

In 1973, a research project was initiated that investigated and developed technology to interlink networks of various kinds together into one "internetwork". This internetwork was held together by protocols that came to be known as the Transmission Control Protocol/Internet Protocol, or TCP/IP, suite, and these protocols stand today as the *de facto* standard for internetworking. Through the 1980s into the 1990s, the roughly 100 protocols in the TCP/IP suite saw many commercial and public domain implementations.

As we saw in Chapters 3 and 5, Symbian OS supports much of the TCP/IP protocol suite. In this chapter, we will look at what it takes to program using TCP/IP on Symbian OS. This is our first look at multipoint connections. Serial and IrDA communications are both point-to-point; TCP/IP protocols address networks that are multipoint in nature. Some of our concepts will adapt to accommodate this new way of communicating.

We will start by discussing some new concepts that arise when we communicate over a network. We will discuss how we connect to a network. The core of this chapter will have a familiar outline: system initialization, opening and configuring TCP/IP sockets, exchanging data, and closing connections. We will also discuss the differences between stream sockets and datagram sockets in TCP/IP. And we will actually get to see more examples than just to-do list item exchanges!

12.1 Considerations for Network Communication

As we have seen, a new medium brings new issues to consider. This section will discuss several new issues with respect to networks.

12.1.1 Connecting to the Network

The network protocol layers we deal with in this chapter all exist above the data link layer. So as we use these protocols, we assume that network connections exist at the data link layer already and that these protocols will interface with whatever exists to connect us to a network.

Figure 12.1 depicts this situation. We are focusing on the top two layers above the horizontal line: IP protocols that make up the network layer and TCP protocols that make up the transport layer. The other layers are not our concern in this chapter but are important for the protocols we *are* concerned about to function. As we covered in Chapter 4, Ethernet, PPP, and IrLAN are protocols that we can use to transport TCP/IP data. These are generically referred to as *network interfaces*.

Symbian OS implements control over these network interfaces through the *network interface manager* or NifMan. While it may sound like some odd superhero, NifMan is actually a server that is used by Symbian OS when a transport is needed, but a network interface has not been started. The server uses a plug-in method for modules to implement such interfaces. We will work with NifMan in the next section.

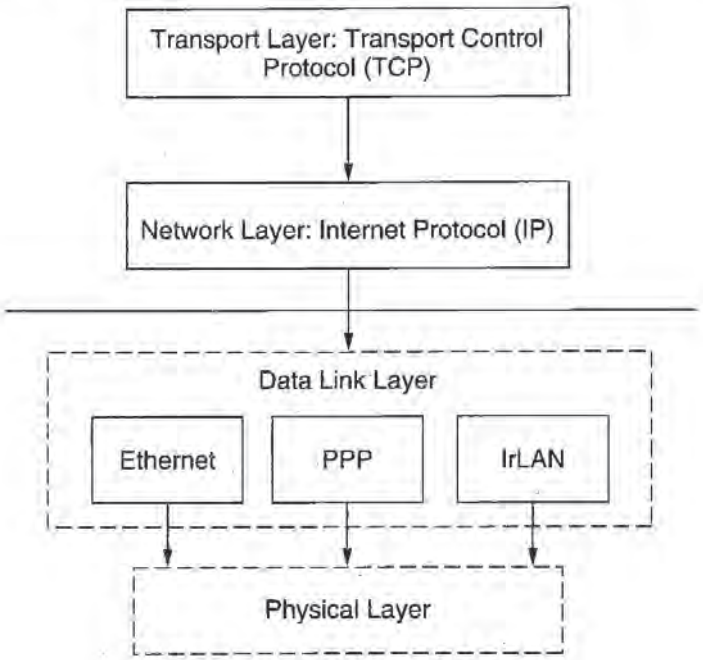

Figure 12.1 Protocol layers in this chapter.

12.1.2 TCP/IP Protocol Review

There are three types of protocols we are concerned with when we program with TCP/IP layers.

- TCP protocols implement a reliable, connected link between two communication endpoints. Any application protocol that desires a reliable connection will probably use TCP as its transport method. For example, HTTP and Telnet are two protocols that typically manage *sessions* of connectivity—long periods of data exchange for which connected links are designed.
- UDP protocols implement a connectionless link between a device and one or more endpoints. By definition, this is an unreliable connection that uses datagrams. Protocols that implement short messages or information queries typically use UDP as their transport method, as do name services such as DNS.
- IP protocols provide the network connections that support TCP and UDP communication. Addressing and routing are major concerns at this layer to identify machines and get data packets from point to point. IP layers on different devices exchange control and routing information with each other (through protocols like Internet Control Message Protocol (ICMP) and Routing Information Protocol (RIP)).

Most TCP protocols are client/server oriented. The HTTP service is a good example of this. Most request-based protocols that fall into the UDP category are also client/server oriented, where the requestor is the client and the information provider is the server. Network Time Protocol (NTP) is a UDP protocol that requests time updates; most average network devices are NTP clients that request time synchronization from time servers across the Internet.

As you might expect, Symbian OS implements TCP/IP protocols through the socket interface. There are standard definitions for TCP sockets and UDP sockets. IP sockets require a bit more ingenuity to use: there are a few standard definitions (e.g., for ICMP implementations), but we need to do some special programming to have complete access to a datagram's format.

12.1.3 System Definitions for TCP/IP

As with IrDA sockets, Symbian OS has set up certain parameters and definitions that we must use for TCP/IP programming.

IP addresses are characterized by the `KAfInet` address family. Specific protocols are given their own constant values (in `\Epoc32\include\in_sock.h`):

```
const TUint KProtocolInetIcmp = 1;
const TUint KProtocolInetTcp = 6;
const TUint KProtocolInetUdp = 17;
const TUint KProtocolInetIp = 0x100;
```

The properties of IP addresses are defined by the `TInetAddr` class, which has many functions to construct, compare, and analyze addresses. Many definitions are based on a single convenience macro:

```
#define INET_ADDR(a,b,c,d) \
   (TUint32)(((a)<<24)|((b)<<16)|((c)<<8)|(d))
```

Since an IP address is really just a 32-bit integer (at least at IP version 4), this macro allows specifying them in the popularized "dotted" notation. We can give `192.168.100.5` as `INET_ADDR(192,168,100,5)`, which will produce the real integer address: `3232261125`.

IP Address Classes

A good way to familiarize yourself with the IP address macro definitions is to try working through an example or two with them.

IP addresses are broken up into address classes, based on which part of the address is the network portion and which is the host portion. A class A address uses the first octet as the network portion, class B uses the first two octets, and class C uses the first three. Go through the macro definitions, and figure out how to extract the network and host parts of an IP address for each of the three address classes.

A look through `\Epoc32\include\in_sock.h` will reveal many options and protocol setting values. We will be discussing these throughout this chapter.

12.2 The Network Interface Manager: NifMan

Because of the modularity of the socket concept, TCP/IP layer socket connections assume a layer that implements network connectivity

lies beneath them. Since there are many ways to connect up a network, Symbian OS uses a NifMan server to coordinate and manage network interfaces. The key aspects of NifMan that we are concerned with here are its pluggable interface and the ability to monitor its progress.

The interface to the NifMan server is used by plug-in modules or *agents*. Agents are implementations of the data link layer for a particular connection method. Agents implement a network interface and give the NifMan server a way to check the progress of the connecting of that interface to the network. The plugin portion of the interface is found in a `.agt` file; the implementation of the network interface itself is found in a `.nif` file.

A good example of an interface agent is the NetDial agent. The NetDial agent allows the NifMan server to check on the progress of a dial-up network connection. The network interface itself is PPP, but the agent interfaces the NifMan server with the kernel so that progress can be checked.

Look for Agents

Look for agents and network interface implementations on your Symbian device or on your emulator.

The emulator has one agent that actual devices do not: `ntras.agt`. This interface implements an agent to monitor the connection between an emulator and the Remote Access Service of a PC running Windows NT. What interface do you think can be used with this agent?

There are also agents on hardware devices that you might not find on emulators. Psion netBooks, for example, might have an agent and a network interface that implement Ethernet support through the PC card slot. Look for agents and network interfaces specifically designed for your device.

The progress indicators from the NifMan server are accessed through an `RNif` class object. Once an `RNif` object is instantiated, you have two choices as to how to get progress reports: synchronously or asynchronously. The functions below access the progress indicators:

```
TInt Progress(TNifProgress &aProgress);
void ProgressNotification(TNifProgressBuf &aProgress,
   TRequestStatus &aStatus);
void CancelProgressNotification();
```

The first function is the synchronous function. The `TNifProgress` object parameter must be filled in by the function. The `TNif-Progress` class is used for accessing information only:

```
class TNifProgress
{
public:
  inline TNifProgress();
  TInt iStage;
  TInt iError;
};
```

Upon successful function completion, the `iStage` variable in the structure will have information pertaining to the connection progress. There are five states a connection can be in; the `TAgent-ConnectType` enumeration in `\Epoc32\include\nifman.h` details the values.

The other two functions are for the asynchronous call—one makes the request and the other cancels the request. The `ProgressNo-tification()` call includes a buffer that holds a progress object and a status variable for monitoring.

There are some other functions in the `RNif` class that are of interest:

- `TInt LastProgressError(`
  `TNifProgress &aProgress)`
  This reports the most recent error to have occurred in the progress of the network interface connection.
- `TInt NetworkActive(TBool &aIsActive)`
  This call will return the status of the network in the included parameter.

The NifMan is useful in a number of ways, typically to manipulate the network interface. Section 12.7 has an example of how to use a NifMan object to check the status of network interface and to shut it down.

12.3 Initializing the System for TCP/IP

Initialization for TCP/IP is much the same as it was for IrDA: connect to the socket server and load the device drivers. This fragment connects the current object to the socket server:

```
result = socksvr.Connect();
User::LeaveIfError(result);
```

This code should look fairly mundane by now.

Because TCP/IP protocols make up a large part of what the socket server does, the protocol modules are loaded by default. You can, of course, load them manually by using the `LoadCommModule()` functions in the `RSocketServ` class, but it is usually unnecessary. The name of the protocol module is "Tcpip".

12.4 Opening and Configuring a TCP/IP Socket

With TCP/IP sockets, opening a socket and configuring a socket are usually done at the same time. Once the initial open (with its configuration parameters) has been completed, the socket requires different configuration options to be applied, depending on the service status of the device as a client or server.

12.4.1 Choosing a Socket Configuration

There are several choices to make about socket configuration; these choices determine what type of socket will be opened. We have discussed these choices before:

- *Stream- vs. datagram-based:* Stream-based sockets are connected sockets, implementing reliable data transfer. Datagram-based sockets are connectionless sockets, implementing unreliable data transfer. For TCP/IP, Symbian OS also supports two other modes: sequential packet transport, which ensures ordered packet arrival but no error checking, and raw transport, which essentially bypasses the protocol layers to send raw data packets through the data link layer, with no guarantees of anything.
- *Type of TCP/IP protocol:* Here, we have the three choices we outlined above: TCP, UDP, or IP. TCP sockets are stream-based, connected sockets. UDP and IP sockets are datagram-based, connectionless sockets.

Opening a TCP/IP socket is done in the standard way—calling `Open()` from the `RSocket` class. The address family is `KAfInet`. The socket type is chosen from the list below:

```
const TUint KsockStream = 1;
const TUint KsockDatagram = 2;
const TUint KSockSeqPacket = 3;
const TUint KSockRaw = 4;
```

Each of these constants corresponds to a transport mode. The socket protocol is chosen from the following list:

```
const TUint KProtocolInetIcmp = 1;
const TUint KProtocolInetTcp = 6;
```

```
const TUint KProtocolInetUdp = 17;
const TUint KProtocolInetIp = 0x100;
```

Note that ICMP is a special protocol supported directly by Symbian OS.

Let's take some examples. If we were to open a socket to a Web server, it would be a TCP-level, connected socket. We could open one with the statement below:

```
result = WebServerSocket.Open(socksvr, KAfInet,
   KSockStream, KProtocolInetTcp);
```

If we wanted to open a socket to a Domain Name Service (DNS) server, we would want a datagram-based socket that worked with UDP protocols. We could open one with the statement below:

```
result = DNSServerSocket.Open(socksvr, KAfInet,
   KSockDatagram, KProtocolInetUdp);
```

If we wanted to open a socket to use ICMP to ping another computer, that would use IP-level protocols, but we would have to assemble our own packets. Hence, we would want to specify raw packets in this way:

```
result = ICMPSocket.Open(socksvr, KAfInet, KSockRaw,
   KProtocolInetIcmp);
```

Other TCP/IP Socket Configurations

The configuration you choose for a socket is really dependent on the type of service you intend on accessing with that socket. As with the three examples above, how the service is offered from a server determines how the socket should be configured. Most standard TCP/IP services have standard configuration requirements.

Consider the services below. For each of these, think about how you would have to configure a socket to access the service.

- POP3 mail servicey
- IMAP mail servicey
- NNTP news reading service
- Telnet remote connection servicey
- FTP file transfer service (FTP is actually two services: a command connection and a data connection)y
- Daytime time update service (not necessarily NTP; try port 13)

12.4.2 Finishing Configuration: Connecting Client Sockets

Client sockets must connect to servers via the `Connect()` function from the `Rsocket` class. Before a socket is connected, however, it must have an address to which it will connect. This is done by constructing a `TInetAddr` object.

Two components make up a `TInetAddr` object: an IP address and a port number. Recall that port numbers are the way specific applications or services are identified on a device and that they are used to tag a datastream so that data connections can be multiplexed on a server. To connect sockets, then, we need to set up both address and port specifications in the `TInetAddr` object, as below:

```
address.SetPort(INET_ADDR(192,168,100,10), 80);
```

This would set the IP address to `192.168.100.10` and the port to `80` (the HTTP service port).

Once the `TInetAddr` object is set, the `Connect()` function can be called, and the socket can attempt to connect to the device and port specified by its arguments:

```
WebServerSocket.Connect(address, status);
```

Remember that `Connect()` is an asynchronous function and requires a status variable to monitor its progress.

12.4.3 Finishing Configuration: Binding Server Sockets

Configuring sockets used by servers means that we must bind the opened socket to a local address, set up a listening queue on the socket, and open a second socket to connect to the incoming connection.

For TCP/IP servers, we must bind the socket to the address and port for which we want to listen. Special settings exist for *any* address or *any* port, but these must be set explicitly in a `TInetAddr` object. For example, the code below will create an address that will match any IP address with its port set to 1100:

```
TInetAddr ServerAddress = new TInetAddr(
   KInetAddrAny, 1100);
```

The constant `KInetPortAny` will match any port. (In reality, these constants are simply a specification of `0.0.0.0` as the IP address and `0` as the port number, respectively.)

Once we have constructed a `TInetAddr` object, binding the server socket is straightforward—it is the same as for IrDA sockets.

Setting up the listening queue is also simple. Consider the following code (where `iListeningSocket` is a `RSocket` object that has already been created):

```
TInetAddr iListeningAddress =
   new TInetAddr(KinetAddrAny, 2000);
iListeningSocket.Bind(iListeningAddress);
iListeningSocket.Listen(5);
```

This code sets up a socket to listen for incoming connections on any IP address looking for port `2000`. The listening queue is set up for five simultaneous connections.

Before a server can accept connections, it must open up a second blank socket to use for an incoming connection.

```
result = iServiceSocket.Open(iSockSvr);
User::LeaveIFError(result);
```

Finally the server can wait for connections by calling the `Accept()` function on the original listener socket:

```
iListener.Accept(iServiceSocket, status);
```

12.4.4 Using the Host Resolver

It is often the case that we have a hostname to connect to or listen for rather than an address. In this case, we must convert this hostname to a `TInetAddr` object prior to using it for a `Connect()` or a `Bind()` call.

The sequence we use looks very much like what we did for IrDA sockets, except there is no real discovery of devices. Instead, the host resolver takes advantage of Domain Name Service (DNS) to do the name-to-address conversion. Consider the following code:

```
RHostResolver resolver;
TNameEntry iHostEntry;
TPtrC AddressName;
TInetAddr Address;
result = resolver.Open(socksvr, KAfInet, KProtocolInetTcp);
User::LeaveIfError(result);
AddressName.Set(_L("http://www.symbian.com"));
resolver.GetByName(AddressName, iHostEntry, status);
User::WaitForRequest(status);
Address = iHostEntry().iAddr;
Address.SetPort(80);

WebServer.Connect(Address, status);
```

Here, `WebServer` is an instantiated `RSocket` object. This code will resolve the host name ***http://www.symbian.com*** to its actual IP address, then combine that address with a specification of port 80 and connect to it on behalf of a client.

Resolver Socket Properties

Stop and think for a moment about the host resolver and the type and protocol of the socket that is opened. How does the call above differ from what was used for IrDA host resolution? Why is the protocol above `KProtocolInetTcp`?

12.4.5 Getting Information from a Socket

After a socket has been opened and configured, and the `Connect()` or `Bind()` and `Accept()` functions have completed, it may be necessary to extract some information from the socket. For example, you may want figure out the address of the remote side of the socket. The `LocalName()` and `RemoteName()` functions handle extracting this information.

```
void LocalName(TSockAddr &anAddr);
void RemoteName(TSockAddr &anAddr);
```

These functions are similar in the way you call them. `LocalName()` returns a `TSockAddr` object that—when converted to a `TInetAddr` object—will give the address and port number of the socket as it is bound in the current application. `RemoteName()` will do the same for the remote endpoint. `LocalName()` is especially useful in situations where you do not know the local address of the device—for example, if it was set up by someone else or by a network service (e.g., a server using Dynamic Host Configuration Protocol (DHCP)). The values of these calls are valid after `Bind()` has been used on servers and `Connect()` has been used on clients.

If just the port information is needed, you can use the functions below:

```
TUint LocalPort();
TUint RemotePort();
```

12.4.6 TCP/IP Socket Options

There are many options available for TCP/IP sockets, much more than I can cover here without putting you to sleep. Most are meant

for specific situations, for example, getting routing information from an ICMP packet, rather than general TCP/IP packet settings. All are set using `Ioctl()` calls. Remember that `Ioctl()` varies its format depending on the option and whether that option is being set or simply queried for its value. The options are given in `\Epoc32\include\in_sock.h`.

12.5 Exchanging Data

Once we have opened and configured a socket and have either connected or accepted a connection, we have a socket with applications on both ends looking to exchange data. At this point, we will use the same API as for other socket connections. There is very little that changes from IrDA socket data exchange to TCP/IP socket data exchange.

We do, however, have access to more flexibility with TCP/IP sockets. We can access raw sockets, for example, that allow the formation of any type of data packet. If we do not want to exchange byte streams, for example, but want to send specific types of data, we need to use raw sockets. An NTP client request, for example, has a specific packet format that must be sent from an IP layer. We can construct this using raw sockets.

Because the API is the same, protocol issues do not change. We still have protocol initialization and termination concerns and must adjust our access to the type of socket connection (streams or datagrams) that we are using.

12.6 Closing a Socket

Closing a TCP/IP socket is no different than closing other sockets: we should drain stream sockets of data requests (if we need to) and then close down the socket. Draining can happen synchronously with the `Cancel()` function or asynchronously with the `Shutdown()` function. Then we call the `Close()` function on the socket to close it down and free up its resources.

12.7 Closing and Checking Network Interfaces

As we stated in Section 12.2, we can access the status of a network interface with an `RNif` object. In addition to simply checking connection progress, we can check the condition of the network interface and can shut it down through objects of this class.

Consider the function below:

```
TBool CPullerAppUi::IsNetworkOpen()
{
    TInt result;
    TBool res;
    RNif iNif;

    // Open the network interface
    result = iNif.Open();
    if (result != KErrNone) return EFalse;

    // Now test it
    res = ETrue;
    result = iNif.NetworkActive(res);
    if (result != KErrNone) res = EFalse;
    iNif.Close();

    return res;
}
```

This function uses an `RNif` object to open a network interface and check the active status through that object.

We can also shutdown an interface using an RNif object. The function below demonstrates this:

```
void CPullerAppUi::DisconnectFromNetwork()
{
    RNif iNif;
    TInt result;

    result = iNif.Open();
    if (result != KErrNone) return EFalse;
    result = iNif.Stop();
    iNif.Close();
}
```

The `Stop()` function will close down a network interface that has been successfully opened.

12.8 Another Example: Fetching a Web Page

Let's consider a new example. Let's say we want an application that will fetch and store a Web page. We will develop a class, `PageGrabber`, whose specification looks something like this:

```
class PageGrabber : public CActive
{
   enum TState {ENotConnected, ENameResolved,
                ESocketOpened, EGrabbingPage,
                EShuttingDownConnection};
```

```
public:
   PageGrabber(CPullerAppUi* aAppUi);
   ~PageGrabber();

public:
   static PageGrabber* NewLC(CPullerAppUi* iAppUi);
   static PageGrabber* NewL(CPullerAppUi* iAppUi);

public:
   void ConstructL();
   void IssueRequest(URL aURL, TDes* DestDir,
      RFs afsSession);

   void DoCancel();
   void RunL();

private:
   CPullerAppUi* iAppUi;
   URL webpage;
   RSocketServ socksvr;
   RSocket sock;
   RHostResolver resolver;
   RFile outstream, logstream;
   RFs fsSession;
   TState iConnectState;
   TNameEntry iHostEnt;
   TRequestStatus iStatus;
   TSockXfrLength reslen;
   TInt count;
};
```

An object of this class is to be created with an `AppUi` class (i.e., an application interface) as an argument. The purpose of this class is to retrieve a Web page and store it in a file. This class is implemented as an active object and is thus derived from the `CActive` class.

12.8.1 Constructors and Deconstructors

We start here with the class constructor.

```
PageGrabber::PageGrabber(CPullerAppUi* aAppUi)
    : CActive()
{
   iAppUi = aAppUi;
   iConnectState = ENotConnected;
   count = 0;
};
```

Notice here that in addition to setting up some variables, we are initializing a state variable `iConnectState` that will be used to control the action of the active object. The initial value reflects the fact that no socket is connected to a remote Web server.

We define a few functions that will allow for the safe creation of `PageGrabber` objects. These call the class contructor and use a clean up stack to safely manipulate memory.

```
PageGrabber* PageGrabber::NewLC(CPullerAppUi* aAppUi)
{
   PageGrabber* self=new (ELeave) PageGrabber(aAppUi);
   CleanupStack::PushL(self);
   self->ConstructL();

   return self;
}

PageGrabber* PageGrabber::NewL(CPullerAppUi* aAppUi)
{
   PageGrabber* self = NewLC(aAppUi);
   CleanupStack::Pop();

   return self;
}
```

The second phase constructor `ConstructL()` will simply add this active object to the scheduling queue.

```
void PageGrabber::ConstructL()
{
   CActiveScheduler::Add(this);
}
```

There is not much to deconstruct. We make sure that the active object is cancelled.

```
PageGrabber::~PageGrabber()
{
   CActive::Cancel();
}
```

12.8.2 Cancelling the Page Grabber

Recall that an active object must have two functions implemented by subclasses: `DoCancel()` and `RunL()`. We will implement `RunL()` in a later section; the implementation of `DoCancel()` is below:

```
void PageGrabber::DoCancel()
{
   sock.CancelAll();
   sock.Close();
   outstream.Close();
}
```

It consists of several `Close()` calls with a synchronous canceling of I/O requests on the socket.

12.8.3 The PageGrabber State Diagram

To understand how pages are grabbed, we must first look at the state diagram for the `PageGrabber` active object. The set of states the active object can be in is given by the following enumeration:

```
enum TState {ENotConnected, ENameResolved, ESocketOpened,
             EGrabbingPage,
             EShuttingDownConnection};
```

There are six possible states, and Figure 12.2 shows how we transition between them. The transitions are labeled with the calls that each state transition is waiting for or the condition that causes the transition. The diagram is a bit simplistic in that transitions to the error state have been left out.

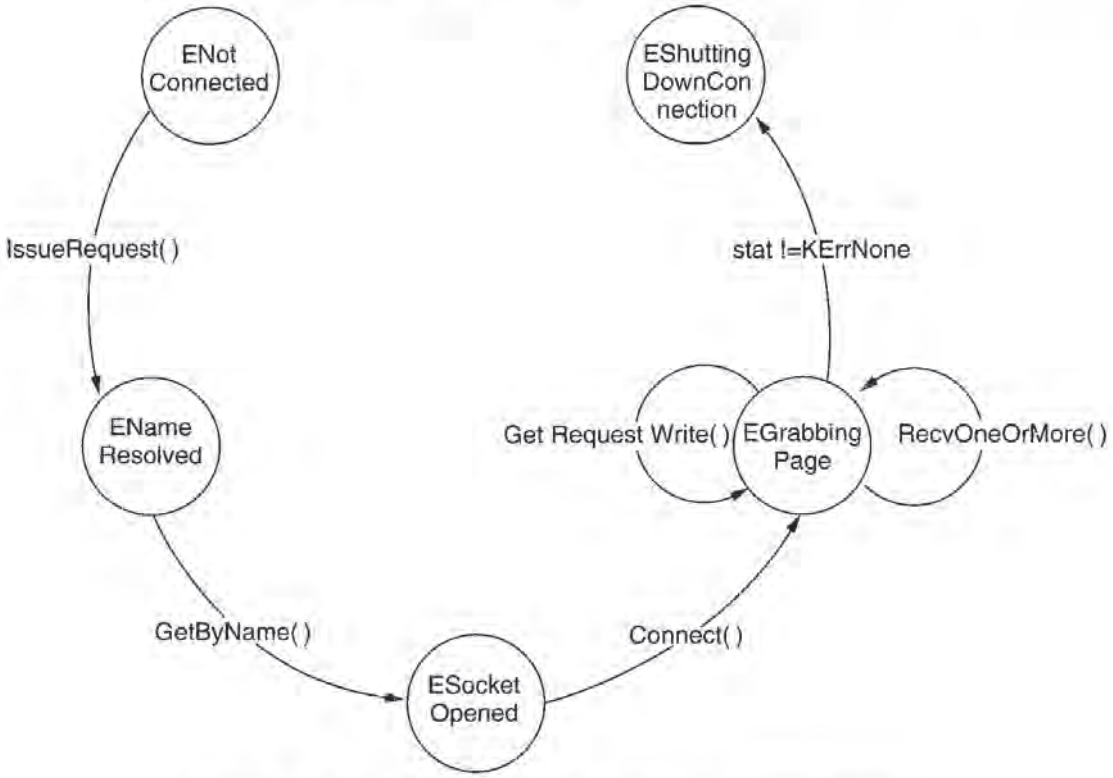

Figure 12.2 State diagram for the PageGrabber active object.

12.8.4 Starting the Page Grab

The grabbing of a page is started by calling the `IssueRequest()` function of the `PageGrabber` class. The calling application will use something like this:

```
iGrabber = PageGrabber::NewL(this);
iGrabber->IssueRequest(webpage, &fname, fsSession);
```

where `iGrabber` is an object of the `PageGrabber` class, `webpage` is defined to be of type `URL`, `fname` (a filename) is a string (`TBuf<256>` is easy), and `fsSession` (a file server session) is of type `RFs`. This call will return right away and will set a page grab in motion.

N.B. The `URL` class is one written for this application. The functions are self-explanatory; please look them up in the source code.

The definition of `IssueRequest()` is below. The code does two things: it generates a random filename if the filename given is formed incorrectly, and it starts the active object sequence.

```
void PageGrabber::IssueRequest(URL aURL, TDes *aFilename,
   RFs afsSession)
{
   TBuf<256> str;
   TBuf<25> nfname;
   TInt err,pos;
   TPtrC iAddressDesc;
   TParse filestorename;

   // Copy the arguments for later use
   fsSession = afsSession;
   webpage.Copy(aURL);

   // Run some tests on the filename given
   fsSession.Parse(*aFilename,filestorename);
   ConeUtils::EnsurePathExistsL(
      filestorename.DriveAndPath());
   err = outstream.Replace(fsSession,
         filestorename.FullName(),
         EFileWrite|EFileStreamText);

   // The filename was a bad name (empty or something else)
   // We generate a random name here
   if (err == KErrBadName) {
      TTime now;
      TInt64 seed = now.Int64();
      now.HomeTime();
      nfname.Format(_L("fn%d"), (TInt)Math::Rand(seed));
```

```
        pos = aFilename->LocateReverse('\\');
        str.Copy(aFilename->Left(pos+1));
        str.Append(nfname);
        str.Append(_L(".htm"));
        fsSession.Parse(str,filestorename);
        err = outstream.Replace(fsSession,
              filestorename.FullName(),
              EFileWrite|EFileStreamText);
    }

    // Abandon ship if there is still a problem
    User::LeaveIfError(err);

    // Announce our intentions
    str.Copy(_L("Opening Network Connection"));
    iAppUi->SetProgressText(&str);

    // Connect to the socket server
    err = socksvr.Connect();
    User::LeaveIfError(err);

    // Now resolve the name given in the URL
    err = resolver.Open(socksvr, KAfInet, KProtocolInetTcp);
    User::LeaveIfError(err);
    iAddressDesc.Set(aURL.host);
    resolver.GetByName(iAddressDesc, iHostEnt, iStatus);

    // This is the first step in the process. Set the
    // connection state and kick the active object
    iConnectState = ENameResolved;
    SetActive();
}
```

This code checks the filename, opens it, and starts the resolver resolving the hostname of the URL given. Note that the host resolver can resolve numeric IP addresses as well as names, which means that a name like `192.168.100.10` will resolve just as well as ***http://www.symbian.com***, and both are legal.

12.8.5 Implementing `RunL()`

We implement `RunL()` as a large `switch` statement. This function will run when pending I/O functions return, and how these returns are handled is based on the value of the `iConnectState` variable, run through the `switch` statement. Let's look at this function, walking through the `switch` statement one case at a time.

The first case is used when name resolution has completed.

```
void PageGrabber::RunL()
{
    TInt pos,err;
```

```
TBuf8<800> line;
TInt portno;

switch (iConnectState) {
   case ENameResolved:
      if (iStatus == KErrNone) {
         // set up the address
         portno = webpage.port;
         address = iHostEnt().iAddr;
         address.SetPort(portno);

         // open the socket
         err = sock.Open(socksvr, KAfInet,
               KSockStream, KProtocolInetTcp);
         User::LeaveIfError(err);

         // start the connection process
         Sock.Connect(address, iStatus);

         // change the state and start the wait
         iConnectState = ESocketOpened;
         SetActive();
      } else {
         // handle an error in name resolution
      }

      break;
```

If name resolution has completed successfully, we should have an address value in the `address` variable. The next step is to retrieve the IP address from the resolution results and the port number from the URL and to build a `TInetAddr` object from them. We then use this object to open a socket and start the connection to the remote Web server.

When the case below is invoked, we have a connection, and the remote Web server is waiting for a command from this process.

```
case ESocketOpened:
   if (iStatus == KErrNone) {
      // Set up the command string for the Web page
      // fetch
      line.Copy(_L("GET "));
      line.Append(webpage.protocol);
      line.Append(_L("://"));
      line.Append(webpage.host);
      line.Append(webpage.file);
      for (pos=0; pos<line.Length(); pos++)
         if (line[pos] == '\\') line[pos] = '/';
      line.Append(_L("\xD\xA"));

      // Send this command to the Web server
      sock->Write(line, iStatus);
      // Move a new state and return to the scheduler
```

```
        iConnectState = EGrabbingPage;
        SetActive();
    } else {
        // handle a connection error
    }

    break;
```

The code constructs a simple GET command and sends it to the Web server (we discussed the GET command in Section 5.2.1). Note the Append() call that puts the string "\xD\xA" at the end of the command string. This is necessary for the Web server to find the end of the string.

The third case is the response from the Web server with the Web page we asked for or a set of HTML lines stating an error. Either way we store it to a file.

```
case EGrabbingPage:
    if (iStatus == KErrNone) {
        // Set up the receive
        reslen = 250;
        sock->RecvOneOrMore(aDesc,0, iStatus,reslen);
        outstream.Write(*aDesc);

        // Record and wait!
        iConnectState = EGrabbingPage;
        iProcessingState = ENormal;
        SetActive();
    } else {
       iConnectState = EShuttingDownConnection;
       // signal connection shutdown
    }

    break;
```

Here, we use RecvOneOrMore() to pick up data from the socket and write whatever we have to the file. We use RecvOneOrMore() to buffer the effects of network delay or disconnection from the network. When an error occurs, either the stream is closed by the remote side or some connection error has occurred. Either way, we shut down the connection.

Finally, this last Boolean condition finds the case where we are shutting down our socket connection.

```
if (iConnectState == EShuttingDownConnection) {
    iAppUi->ShutDownGrab(); // signal shutdown to app
    sock.CancelAll();
    sock.Close();
    outstream.Close();
    delete this;
}
```

In this code, we signal to the application that the connection is down, and we go through the standard socket shutdown routine. Notice that at the end of the code, the active object deletes itself. This is an easy way to make sure the object is deleted only after all the socket I/O has been completed.

Deleting the Socket

In the code above, the active object deletes itself. It is slightly odd behavior for an object to delete itself. How could you structure it so that the object that created this active object correctly deleted it? What would you have to check in order to make sure all resources were correctly handled? What issues are involved?

Further, can you tell why we allocated a socket as a pointer in the first place? Why did we not just use a declared socket object?

12.9 Sending Datagrams

Protocols such as HTTP, FTP, Telnet, and WAP work with connected sockets with data streams. However, many UDP and IP connections need data sent in different ways. By their nature, these protocols are designed in request-response pairs, and the request must have a specific format.

Consider, as an example, Network Time Protocol (NTP). NTP servers are supposed to have access to accurate time sources and to respond with a sample of the accurate time when an NTP client makes a request. This request comes in on port 123 and takes the form of an NTP packet, shown in Figure 12.3. NTP socket communication is a UDP connection using datagrams.

At first glance, sending NTP packets may appear straightforward. One simple way to do this would be to use multiple `Write()` or `SendTo()` statements, one per field in the NTP packet. This involves several mistakes, but the biggest is that each statement causes data to be sent out as a datagram, resulting in the NTP case as 15 datagrams, each in the wrong format, rather than a single, correct datagram.

A correct way to send datagrams with specific formats is to send a single buffer containing all the data in the datagram. It can be a bit challenging to configure a buffer full of bytes in the correct format, but C++ helps us out through the ability to overlay memory.

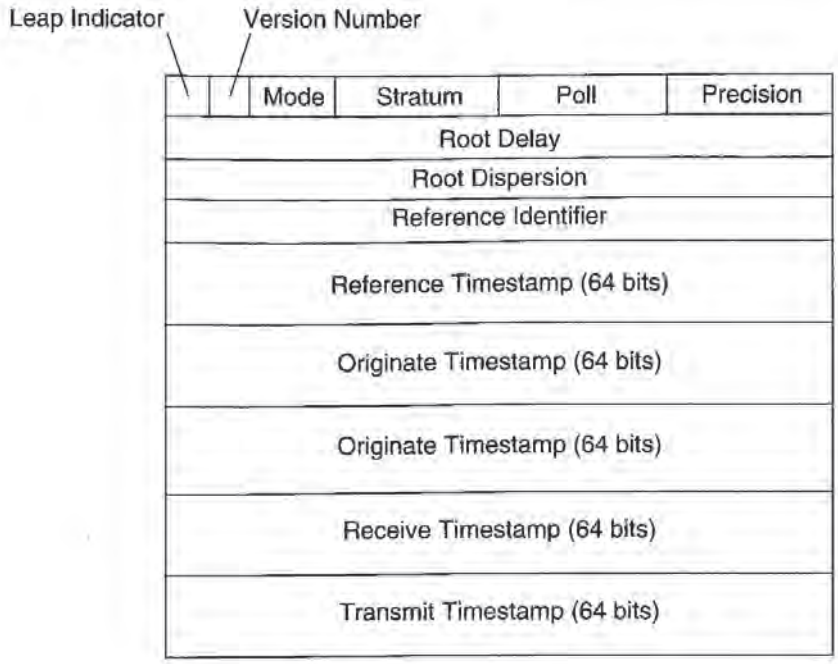

Figure 12.3 NTP datagram format.

Let's consider a C++ structure that depicts an NTP datagram:

```
struct pkt {
   u_char li_vn_mode;  /* contains leap indicator, version
                          and mode */
   u_char stratum;     /* peer's stratum */
   u_char ppoll;  /* the peer polling interval */
   s_char precision;   /* peer clock precision */
   s_fp rootdelay;     /* distance to primary clock */
   u_fp rootdispersion;/* clock dispersion */
   u_int32 refid;      /* reference clock ID */
   l_fp reftime;  /* time peer clock was last updated */
   l_fp org;  /* originate time stamp */
   l_fp rec;  /* receive time stamp */
   l_fp xmt;  /* transmit time stamp */
};
```

The data types referenced above are not standard Symbian types. This code is used from public domain source code for NTP services. Symbian OS design adapts well to this code as well as its own. The types are as follows:

- `u_char`: unsigned char, 8 bits
- `s_char`: signed char, 8 bits
- `u_int32`: unsigned integer, 32 bits
- `u_fp`: unsigned floating point, 32 bits

- `s_fp`: signed floating point, 32 bits
- `l_fp`: long, signed floating point: 64 bits

To build a buffer that holds this packet, we declare the following:

```
u_char outpack[64];
struct pkt *xpkt = (struct pkt *)outpack;
TBuf8<64> icpbuffer;
```

Here, we have declared a buffer of 8-bit bytes, 64 bytes long. Through the magic of C++, we have made the memory space of this buffer occupy the same memory space as the `pkt` structure. So any changes we make to the memory referenced by the `xpkt` pointer will be made in the `outpack` buffer. When we are finished constructing the NTP packet, we copy the `outpack` buffer to the `icpbuffer` and send the `icpbuffer` to NTP server.

We can see this in the portion of the `RunL()` function shown below. In this code, we have already connected a socket to the remote NTP server and must now construct the packet and send it. There is a fair amount of the NTP client code I am not showing you—so that we can illustrate sending a datagram packet—so bear with some of the macro definitions.

```
case ESocketOpened:
   if (iStatus == KErrNone) {
      // assemble the NTP packet
        xpkt->li_vn_mode = PKT_LI_VN_MODE(LEAP_NOTINSYNC,
                           sys_version, MODE_CLIENT);
     xpkt->stratum = STRATUM_TO_PKT(STRATUM_UNSPEC);
     xpkt->ppoll = NTP_MINPOLL;
     xpkt->precision = NTPDATE_PRECISION;
     xpkt->rootdelay = htonl(NTPDATE_DISTANCE);
     xpkt->rootdispersion = htonl(NTPDATE_DISP);
     xpkt->refid = htonl(NTPDATE_REFID);
     // Clear the time fields
     L_CLR(&xpkt->reftime);
     L_CLR(&xpkt->org);
     L_CLR(&xpkt->rec);

      // get the local time and put it into the
      // time fields
      l_fp xmt;
      (void) gettimeofday(&tv, (struct timezone *)0);
      xmt.l_i = tv.tv_sec + JAN_1970;
      dtemp = tv.tv_usec * FRAC / 1e6;
      if (dtemp >= FRAC) xmt.l_i++;
      xmt.l_uf = (u_int32)dtemp;
      xpkt->xmt.l_ui = htonl(xmt.l_ui);
      xpkt->xmt.l_uf = htonl(xmt.l_uf);
```

```
        // Send the packet
        icpbuffer.SetLength(0);
        icpbuffer.Append(outpack, MAXPACKET);
        sock.Write(icpbuffer, stat);

        iConnectState = ESendingNTPRequest;
        SetActive();
    } else {
        // signal a connection failure
    }

    break;
```

Notice how the code treats the structure pointed to by the `xpkt` variable as a C++ struct, then copies the same memory space by using the `outpack` buffer as an array.

Other datagram protocols can be worked with in a similar way. These include lookup services like DNS and LDAP, information protocols like RIP (for routing) and WHO (logged in users on a time-sharing system), and miscellaneous protocols like daytime and echo protocols. ICMP would also be accessed the same way, although you need to open a socket of the protocol `KProtocolInetIcmp` and a type of `KSockRaw`.

12.10 Summary

This chapter has surveyed what it takes to communicate using Symbian devices over TCP/IP networks. We looked at several different ways sockets are used to send packets over the different TCP/IP protocol layers. Specifically, we looked at the following topics:

- We briefly reviewed the relationship between TCP/IP and Symbian devices. We reviewed how connections to a network are made, what TCP/IP protocols are supported, and what constructs Symbian OS provides for us to talk about TCP/IP.
- We looked at the NifMan—the network interface manager—and how network interfaces are manipulated.
- We then went through the stages of using sockets and how TCP/IP fits into these stages.
- We reviewed two examples: retrieving a Web page from a server as an example of TCP level protocols and implementing an NTP client as an example of using datagram packets.

The next chapter takes sockets and tackles Bluetooth protocols with them. We will be introducing new examples as well as revisiting our old friend, `TodoX`.

13

Bluetooth Communications

In 908, Thyre, the wife of Denmark's King Gorm the Old, gave birth to a son. The boy was named Harald Blatant—or, translated, Harald Bluetooth—and he became king of Denmark from 940 to 981. He possessed great skills of communication and could make people talk to each other; during his rule, Denmark and Norway were Christianized and united. It is these skills in communication and unification that Bluetooth designers tried to capture as they chose his name to identify the wireless technology that we will examine in this chapter.

Bluetooth identifies a suite of protocols and a model for radio transmission that focuses on short range wireless communication. As with other protocol stacks, the Symbian OS provides support for the Bluetooth protocol suite by extending the socket model to include Bluetooth protocol modules. In addition to Bluetooth sockets, the Symbian OS has a solid implementation of Bluetooth device and service discovery and of the Bluetooth security model.

We will look at this Bluetooth support in this chapter. We will start, as we have with previous chapters, by reviewing the Bluetooth model and the way the Symbian OS addresses it. We will then take a close look at device and service discovery, which features prominently in the way Bluetooth works. We will follow this by relating sockets to the Bluetooth model, through all the socket phases we have discussed in other chapters. We will close the chapter by looking at an example of instant messaging using Bluetooth protocols.

13.1 Considerations for Bluetooth Communication

13.1.1 Reviewing the Bluetooth Model

The Bluetooth model for communications is built from a specification for radio transmission, a driver for this radio module, and a

stack of protocols built on top of the driver that implements various forms of data transmission. Back in Section 4.5, we overviewed the Bluetooth protocols and, in Figure 4.6, we displayed a diagram of these protocols and how they relate to one another.

For our purposes—i.e., transferring data over Bluetooth—we can group the protocols into software and hardware groups and focus on the software. Figure 13.1 holds a redrawing of Figure 4.6, with a separation of protocols into hardware and software groups. LMP, Baseband, and Audio protocols are typically handled by Bluetooth hardware; L2CAP, SDP, and RFCOMM are typically implemented in software.

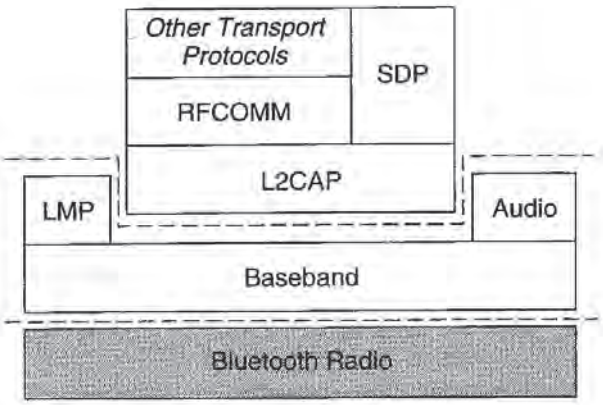

Figure 13.1 Protocol layers in this chapter.

We will focus on the software protocols. We will use sockets that work through the L2CAP layer or the RFCOMM layer. L2CAP sockets allow finer-grained control of the connection, implementing properties much like TCP in the internetwork world. RFCOMM sockets have properties like serial communication. The typical choice of applications is to use L2CAP sockets. SDP will be used to discover remote devices and the services they offer. In the Bluetooth world, discovery is important. As Bluetooth devices are generally mobile, and a Bluetooth radio signal has a short range, it is likely that Bluetooth devices will frequently move in and out of range of other Bluetooth devices. This movement means that discovery may have to be ongoing, an asynchronous activity that proceeds in parallel with an application's functionality.

13.1.2 Spotlight on Bluetooth Device Services

In the other socket-based communications we have covered—IrDA and TCP/IP—we have encountered a two-part addressing of socket connections. In these previous two contexts, addresses were made up of hardware addresses and service ports. For IrDA, the assignment

of service port numbers is arbitrary; for TCP/IP, service port numbers are assigned for some services and arbitrary for others. For IrDA, there were no attributes attached to the port; for TCP/IP, there was a small set of attributes (e.g., protocol and socket type).

The Bluetooth model extends this device/service address pairing for socket connections, taking service address and attribute assignment to a higher level of formality and complexity. Bluetooth services are identified by a unique ID number, and are characterized by a list of service attributes. In turn, each attribute is characterized by an ID number, an attribute type and a value. The set of triples (ID, type, value) that describe a service are stored in a database maintained on each Bluetooth device, with a single record for each service and its attributes. So each Bluetooth device must maintain a database of services, composed of service records, where each record is a list of service attributes.

For example, let's say we have a digital camera that provides an "image" service. We want it to be the source for images for wireless devices. Simply allowing incoming connections to know that it is providing an image service is not enough; devices that might use this service would want to know more information. For example, useful information here might be image format, image height, image width, and color depth. In addition, a remote device might want to determine if the camera also provides a "shutter" service—that is, the ability to take a picture. We might visualize these services like the diagram in Figure 13.2. The database is

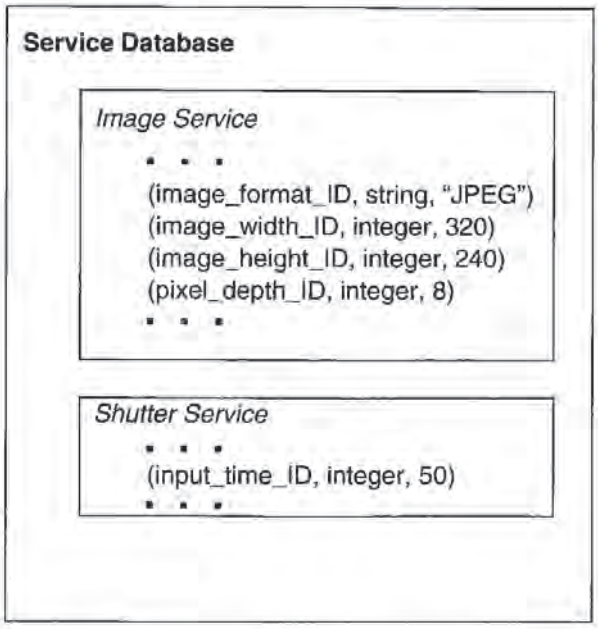

Figure 13.2 Example service database contents.

composed of a set of services, which in turn is a set of attribute triples. In our example, we might have at least four attribute triples for the "image" service.

Bluetooth device discovery, then, can also include service discovery. With Bluetooth, a device not only looks for another Bluetooth device, it might also look for a Bluetooth device that provides a specific service. In the camera example, a handheld Bluetooth device might want to be able to snap pictures that are in JPEG format, of any size, with 8-bit color depth. The camera we described above would match that device search.

It is worth noting here that this type of service discovery is a marked improvement from IrDA and TCP/IP discovery methods. Using IrDA sockets, we are able to find a device and to query its IAS about service attributes. However, this service attribute information is very spare; we can only find port number information or the level of support for various protocols. Using TCP/IP sockets, we are not even able to collect this much information. We must know ahead of time which IP address and which port number we want to connect to; there is no way to ask a machine what ports are open. (In all fairness, there are certainly methods of using TCP/IP to find devices that provide services listening on ports. We could broadcast for devices, for example, or perform a scan on a particular device for open ports. However, this is not part of the socket connection process; this would be a separate, time-consuming operation performed before attempting socket connection.)

13.1.3 System Definitions for Bluetooth

As we have seen with previous protocol stacks, the Symbian OS has defined some system parameters that we must use when programming for Bluetooth communication.

Bluetooth addresses are from the `KBTAddrFamily` address family. System definitions can be found in `\Epoc32\include\bt_sock.h`, where Bluetooth addresses are characterized by the `SBTAddr` structure, as below:

```
struct SBTAddr
{
  TBTDevAddr iAddress;
};
```

In turn, the `TBTDevAddr` class is defined in the file `\Epoc32\include\bttypes.h` as follows:

```
static const TInt KSdpUUIDMaxLength = 16;
const TInt KBTDevAddrSize=0x06;
const TInt KBTMaxDevAddrSize = KBTDevAddrSize;

class TBTDevAddr
{
public:
   IMPORT_C TBTDevAddr();
   IMPORT_C TBTDevAddr(const TInt64 &aInt);
   IMPORT_C TBTDevAddr(const TDesC8 &aDes);
   inline TBool operator==(const TBTDevAddr &aAddr) const;
   inline TBool operator!=(const TBTDevAddr &aAddr) const;
   inline const TUint8 &operator[](TInt aIndex) const;
   inline TUint8 &operator[](TInt aIndex);
   inline void Reset();
   inline TPtr8 Des();
   inline const TPtrC8 Des() const;
private:
   TFixedArray<TUint8, KBTDevAddrSize> iAddr;
};
```

From these definitions, you can see that a Bluetooth address is represented by an array of size `KBTDevAddrSize` (currently six) that contains 8-bit bytes. This makes sense, since Bluetooth addresses are 48-bit quantities.

In the `bt_sock.h` file, there are four protocol definitions:

```
const TUint KBTLinkManager = 0x0099;
const TUint KL2CAP = 0x0100;
const TUint KRFCOMM = 0x0003;
const TUint KSDP = 0x0001;
```

These are for the link manager protocol, L2CAP, RFCOMM, and service discovery protocol, respectively.

Read On

There are many definitions in the `bt_sock.h` and `bttypes.h` files. You should read through these simply to familiarize yourself with the various socket options and error constants. We will address some of these later in this chapter.

13.2 Discovering Bluetooth Devices and Services

As we mentioned in the above discussion about services, device and service discovery are a very important part of using Bluetooth

protocols. In typical Bluetooth use scenarios, devices go in and out of range easily and quickly. Discovery happens regularly and is the best way to monitor this movement and to stay connected to services.

13.2.1 Explicitly Setting the Device and Service

It should be stated first that, while it can be very useful, the discovery of devices and services is not always necessary. A device could have a specific remote device for which it is waiting to exchange data, or the user could specifically request a particular Bluetooth device and service. In these cases, we can set the device address or the service ID explicitly and not bother to do device discovery at all.

Bluetooth device addresses are 48-bit quantities. We need a 64 bit integer to hold them, or a `TInt64`. We use this `TInt64` quantity to set the device address. As an example, if we are exchanging data with a device whose hardware address is `65097113885`, or `0xf2817211d`, we can create a device address explicitly using the following declaration:

```
TBTDevAddr deviceAddress(TInt64(0xf2, 0x817211d));
```

This would construct the device address `deviceAddress` to have the correct predefined setting.

The service can be explicitly set as well. Services in Bluetooth are specified by service ID numbers. If, for example, you choose ID number 100 as the service to exchange data across, you can set this just as you would with IrDA or TCP/IP:

```
TBTSockAddr addr;

addr.SetBTAddr(deviceAddress);
addr.SetPort(100);
```

We could now use the `addr` variable in a `Connect()` method call to use Bluetooth sockets.

13.2.2 Discovering a Device

The sequence used in the Symbian OS for discovering devices looks very much like the one we used for IrDA device discovery: we will use the host resolver to find the devices we need. We can discover devices by a device *address* or by a device *name*.

To discover a device by address, we need to execute code such as the following:

```
TInquirySockAddr inqaddr;
TNameEntry tnentry;

inqaddr.SetIAC(KGIAC);
inqaddr.SetAction(KHostResInquiry);
resolver.GetByAddress(inqaddr, tnentry, status);
User::WaitForRequest(status);
```

The inquiry proceeds as follows. First, we must set the *inquiry access code*; it can be one of the following constants:

```
const TUint KGIAC = 0x9e8b33;
const TUint KLIAC = 0x9e8b00;
```

In the first case, the code signifies a general unlimited query; in the second case, the query is a limited one. A general query requires that all discoverable devices must respond to the discovery query (a broadcast). A limited query forces remote devices to respond only to certain messages (a multicast). In a dense environment like a conference, airport terminal, or classroom, there may be many devices that are discoverable. Regulating the query mechanism is a way to fine-tune the discovery process and reduce discovery time.

For the next step, we set the type of inquiry we are trying to make. The constants are defined as below, for both by-address and by-name discovery:

```
const static TUint KHostResInquiry = 1;
const static TUint KHostResName = 2;
```

Finally, we make the call to the `GetByAddress()` method and wait for a response. This call is asynchronous. Both by-address and by-name discovery are implemented through the `GetByAddress()` method; the `GetByName()` method is invalid for Bluetooth. If the call to `GetByAddress()` succeeds, the response is much like the response for IrDA: the discovered device information will be returned in the `TNameEntry` object that was sent as a parameter. If more than one device is discovered, the information on other devices can be retrieved by use of the `RHostResolver::Next()` method.

13.2.3 Discovering Services

Service discovery is needed when an application does not know ahead of time that a particular remote device has the service it requires. Service discovery is done in two steps: first we search

for the service ID, then we search for the service attributes of the discovered service.

The service discovery agent

Service discovery is done using the Service Discovery Agent (SDA). The agent uses the Bluetooth Service Discovery Protocol (SDP) to search for and match a service on the remote device. SDP is implemented as an asynchronous activity, and the application that started the service search must set up notification callback methods to be notified about the success or failure of the search.

The agent is represented in the Symbian OS by an object of the `CSdpAgent` class. An object of this class encapsulates the use of SDP between remote devices. The `CSdpAgent` class has the following definition:

```
class CSdpAgent : public CBase
{
public:
  IMPORT_C static CSdpAgent* NewL(
     MSdpAgentNotifier &aNotifier,
     const TBTDevAddr &aDevAddr);
  IMPORT_C static CSdpAgent* NewLC(
     MSdpAgentNotifier &aNotifier,
     const TBTDevAddr &aDevAddr);
  IMPORT_C ~CSdpAgent();
  IMPORT_C void SetRecordFilterL(
     const CSdpSearchPattern &aUUIDFilter);
  IMPORT_C void SetAttributePredictorListL(
     const CSdpAttrIdMatchList &aMatchList);
  IMPORT_C void NextRecordRequestL();

  IMPORT_C void AttributeRequestL(
                  TSdpServRecordHandle aHandle,
                  TSdpAttributeID aAttrID);
  IMPORT_C void AttributeRequestL(
                  TSdpServRecordHandle aHandle,
                  const CSdpAttrIdMatchList &aMatchList);
  IMPORT_C void AttributeRequestL(
                  MSdpElementBuilder* aBuilder,
                  TSdpServRecordHandle aHandle,
                  TSdpAttributeID aAttrID);
  IMPORT_C void AttributeRequestL(
                  MSdpElementBuilder* aBuilder,
                  TSdpServRecordHandle aHandle,
                  const CSdpAttrIdMatchList &aMatchList);
     ...
};
```

To start with, an agent needs an `MSdpAgentNotifier` to notify of its progress. This is an object that is defined with specific callbacks:

```
class MSdpAgentNotifier
{
public:
  virtual void NextRecordRequestComplete(TInt aError,
     TSdpServRecordHandle aHandle,
     TInt aTotalRecordsCount) = 0;
  virtual void AttributeRequestResult(
     TSdpServRecordHandle aHandle,TSdpAttributeID aAttrID,
     CSdpAttrValue* aAttrValue) = 0;
  virtual void AttributeRequestComplete(
     TSdpServRecordHandle, TInt aError) = 0;
};
```

The notifier is sent data when search operations are executed. The `NextRecordRequestComplete()` method will be called when the latest request for a matching service search is complete. The call will indicate the error status of the search in the first parameter, give a handle to the service record in the second, and give the number of service records matched in the third. The handle can then be used to request attributes from the database record, and these results are returned by `AttributeRequestResult()` and the completion status by `AttributeRequestComplete()`.

Searching for service IDs

The agent uses search pattern objects to find service IDs. Objects of the `CSdpSearchPattern` class are used to specify the search. This class has the following definition:

```
class CSdpSearchPattern : public CBase, public
MSdpElementBuilder
  {
public:
  IMPORT_C static CSdpSearchPattern* NewL();
  IMPORT_C void ConstructL();
  IMPORT_C ~CSdpSearchPattern();
  IMPORT_C TInt AddL(const TUUID &aUUID);
  IMPORT_C TInt Remove(const TUUID &aUUID);
  IMPORT_C TInt Find(const TUUID &aUUID, TInt &aPos) const;
  IMPORT_C TInt Count() const;
  IMPORT_C const TUUID At(TInt anIndex) const;
  IMPORT_C void Reset();
  inline TBool IsEmpty();
      ...
};
```

Service IDs are added or removed from the search pattern by methods of this class. The IDs are given by the class `TUUID`. The definition of the `TUUID` class enables many different forms of UUID specification; the use of integers is one way to give them.

To use an agent, we need to go through the following steps:

- First we need to create a notifier. This is a user-defined class derived from the `MSdpAgentNotifier` class that defines the method we discussed above. For example, consider the definition below:

```
class AGENTNotifier: public MSdpAgentNotifier
{
   AGENTNotifier();
   ~AGENTNotifier();
   void NextRecordRequestComplete(TInt aError,
      TSdpServRecordHandle aHandle,
      TInt aTotalRecordsCount);
   void AttributeRequestResult(
      TSdpServRecordHandle aHandle,
      TSdpAttributeID aAttrID,
      CSdpAttrValue* aAttrValue);
   void AttributeRequestComplete(TSdpServRecordHandle,
      TInt aError);
}
```

  We must define each of the `AGENTNotifier` methods to activate something in our main application when discovery is complete.

- Next, we need to create the agent. As an example, consider the code below:

```
AGENTNotifier notifier;
CSdpAgent* agent = CSdpAgent::NewLC(notifer,
   deviceAddress);
```

  The device address in `deviceAddress` was constructed as we did it previously—either by explicit declaration or by discovery. This creates an agent that will discover services on the device whose address is specified in `deviceAddress` and will communicate back using the `notifier` object.

- As a third step, we need to create a search pattern and add the service definitions for which we need to look. We need an object of the class `CSdpSearchPattern` to which we add service IDs. The fragment below illustrates this:

```
CSdpSearchPattern* pattern = CSdpSearchPattern::NewL();
pattern->AddL(100);
```

  Here, we have added the service whose ID is `100` to the pattern of services for which we want to look.

- Next, we install the search pattern in the agent. An example is below:

```
agent->SetRecordFilterL(*pattern);
```

- Finally, we signal the agent to start the search process. This results in the active object that defines the agent to asynchronously search the device specified for the service given in the pattern specification. The code below would start this process:

```
agent->NextRecordRequestL();
```

When this method returns, all further communication from the agent is done through the notifier via the callback methods.

When multiple service IDs are added to a search pattern, the effect is like combining these IDs together by using a Boolean OR operator. Therefore, let's say we used the following code to construct a pattern:

```
CSdpSearchPattern* pattern2 = CSdpSearchPattern::NewL();
pattern2->AddL(100);
pattern2->AddL(150);
pattern2->AddL(730);
```

The effect is to provide a pattern that will match a service whose ID is *either* 100, 150, or 730. Once a service has been selected, the application needs to further examine its ID or attributes to determine if that service is to be used.

Note that I have taken some liberties in arbitrarily assigning service ID numbers. In many cases, for established services, the ID numbers are preassigned by the Bluetooth Consortium.

Searching for service attributes

Once a service handle has been successfully obtained through a call to an `MSdpAgentNotifier` class object's `NextRecordRequestComplete()` method, we still have to make requests to the remote database to obtain the service's attributes. This time, we need a pattern specified as an object of the `CSdpAttrIdMatchList` class. The definition of this class is below:

```
class CSdpAttrIdMatchList : public CBase, public
MSdpElementBuilder
{
public:
  IMPORT_C static CSdpAttrIdMatchList* NewL();
```

```
  IMPORT_C static CSdpAttrIdMatchList* NewL(
     const CSdpAttrIdMatchList &aAttrMatchList);
  IMPORT_C ~CSdpAttrIdMatchList();
  IMPORT_C void AddL(TAttrRange aRange);
  IMPORT_C void RemoveL(TAttrRange aRange);
  TInt Find(TSdpAttributeID aAttrId, TInt &aPos) const;
  IMPORT_C TBool InMatchList(TSdpAttributeID aAttrId,
     TInt &aPos) const;
  inline TBool InMatchList(TSdpAttributeID aAttrId) const;
  inline TInt Count() const;
  void FindAttributesL(CSdpServRecord &aRec,
     MAttributeMatchHandler &aHandler) const;
  IMPORT_C TUint EncodeL(CElementEncoder &aEncoder) const;
...
};
```

Notice that this class has many of the same methods that we found for service ID matching—`AddL()`, `RemoveL()`, and `Find()`, for instance—and this means that we build an attribute pattern in much the same way. Notice that `AddL()` and `RemoveL()` take a `TAddrRange` structure as their parameter. This is so that we can specify ranges of attribute IDs rather than one at a time. The `InMatchList()` method is used to see if an attribute ID is already in the pattern, and the `Count()` method will return the number of patterns in an object of this type.

To search for attributes, we set up an attribute ID pattern much the same way as we did for service ID patterns. Now, however, we need to know what service attribute IDs we are looking for. We proceed as follows:

- First, note what we already have: we have an agent (we used it to do the service ID search in the first place), the notifier object the agent uses to communicate, and the service handle that was returned with the callback to the notifier's `NextRecordRequestComplete()` method that signifies the service that was found.
- Now, we need to create a `CSdpAttrIdMatchList` object. The code below shows an example:

```
CSdpAttrIdMatchList* attribute_pattern =
   CSdpAttrIdMatchList::NewL();
```

- Like with service ID discovery, we add the attribute IDs to the attribute pattern. For example, consider this statement

```
attribute_pattern->AddL(TAttrRange(0x102));
```

- Now, we can start the attribute search by using the `AttributeRequest()` method in the agent object. As an example, consider the line below:

```
agent->AttributeRequestL(serviceHandle,
   *attribute_pattern);
```

At this point, the agent releases the caller and runs asynchronously. It communicates again through the notifier, this time using the `AttributeRequestResult()` and `AttributeRequestComplete()` methods in the notifier. For every attribute found, the `AttributeRequestResult()` method is called. When there are no more attributes that match the pattern specified, the `AttributeRequestComplete()` method is called.

13.2.4 The Service Discovery Database

In the previous section, we looked at how to search a remote device's service database. Now we will look at this from the local perspective: how do we get records into the local database to begin with? For example, how would an application register services that it is offering on the local device?

We will look at this in three parts: dealing with the database itself, characterizing service attributes, and working with service database records.

Dealing with the database

Before we can manipulate records in the service database, we have to establish a connection to it. As with most resources in the Symbian OS, the Bluetooth service database is managed by a server. So, as we have before, we first connect with the server. Then we can open and manipulate the database and finish by closing both the database and the server connection.

The database server API is found in the `RSdp` class. Its definition is below:

```
class RSdp : public RSessionBase
{
public:
  IMPORT_C RSdp();
  IMPORT_C TInt Connect();
  IMPORT_C TVersion Version() const;
  IMPORT_C void ResourceCountMarkStart();
  IMPORT_C void ResourceCountMarkEnd();
  IMPORT_C TInt ResourceCount();
};
```

Here, we would use the `Connect()` method to open a server connection. We would stop this connection with the `Close()` method inherited from the `RSessionBase` class. We can get its version number from the `Version()` method (by build number, or by major and minor version numbers). The number of connections to the database server can be counted by using the `ResourceCount...()` group of methods. To start counting, we would use the `ResourceCountMarkStart()` method and, to end the counting, we would use the `ResourceCountMarkEnd()` method. Between these method calls, a call to `ResourceCount()` will return the number of connections to the database. This resource count is a count of open subsessions, that is, the number of applications opening and manipulating the service database at the moment a call is made. This count can be used to monitor the load on a device, for example, or simply to log database use.

The database itself is characterized by the `RSdpDatabase` class. The definition for this class is below:

```
class RSdpDatabase : public RSdpSubSession
{
public:
  IMPORT_C RSdpDatabase();
  IMPORT_C TInt Open(RSdp &aSession);
  IMPORT_C void Close();
  IMPORT_C void CreateServiceRecordL(const TUUID &aUUID,
     TSdpServRecordHandle &aHandle);
  IMPORT_C void CreateServiceRecordL(
     CSdpAttrValueDES &aUUIDList, TSdpServRecordHandle &aHandle);
  IMPORT_C void UpdateAttributeL(TSdpServRecordHandle aHandle,
     TSdpAttributeID aAttrID, CSdpAttrValue &aAttrValue);
  IMPORT_C void UpdateAttributeL(TSdpServRecordHandle aHandle,
     TSdpAttributeID aAttrID, TUint aUintValue);
  IMPORT_C void UpdateAttributeL(TSdpServRecordHandle aHandle,
     TSdpAttributeID aAttrID, const TDesC16 &aDesCValue);
  IMPORT_C void UpdateAttributeL(TSdpServRecordHandle aHandle,
     TSdpAttributeID aAttrID, const TDesC8 &aDesCValue);
  IMPORT_C void DeleteAttributeL(TSdpServRecordHandle aHandle,
     TSdpAttributeID aAttrID);
  IMPORT_C void DeleteRecordL(TSdpServRecordHandle aHandle);
   ...
};
```

There is a lot of declarative jumble in that definition; the core methods are as follows:

- The `Open()` method opens a database, and the `Close()` method closes it down.
- The `CreateServiceRecordL()` methods create service records in the database. The `DeleteRecordL()` method removes service records from the database.

- The `UpdateAttributeL()` methods update attributes in a service record. The `DeleteAttributeL()` method removes attributes from a service record.

The code fragment below illustrates how we might connect to the database server, open the database, and then shut things down again:

```
RSdp database_svr;
RSdpDatabase database;
TInt result;

result = database_svr.Connect();
User::LeaveIfError(result);

result = database.Open(database_svr);
User::LeaveIfError(result);

...  // use the database
database.Close();
database_svr.Close();
```

Characterizing service attributes

Service attributes are complicated beasties. They are easily visualized (see Figure 13.2), but the open flexibility and typing requirements make the API a little cumbersome. However, the Symbian OS designers have taken care to make the API as usable as possible.

An attribute's value can be one of nine possible types. Seven of these are easily represented: integer, unsigned integer, Boolean, URL, string, UUID, and a nil type (used to represent an "undefined" value). The remaining two types are collections of values. A data element alternative (DEA) collection is a list of elements from which a single element will be selected (like a reference to an item in an array). A data element sequence (DES) collection is a list of elements that will be used as a list.

Each data element type is characterized by its own class. These classes are all derived from a base class: the `CSdpAttrValue` class. Figure 13.3 lists the classes and shows what each represents. Let's deal with these in two parts: the "simple" data types and the element collections.

Simple attribute types Rather than examining each individual class, we will look at the `CSdpAttrValueInt` class as a representative of the others. The class definition is given below:

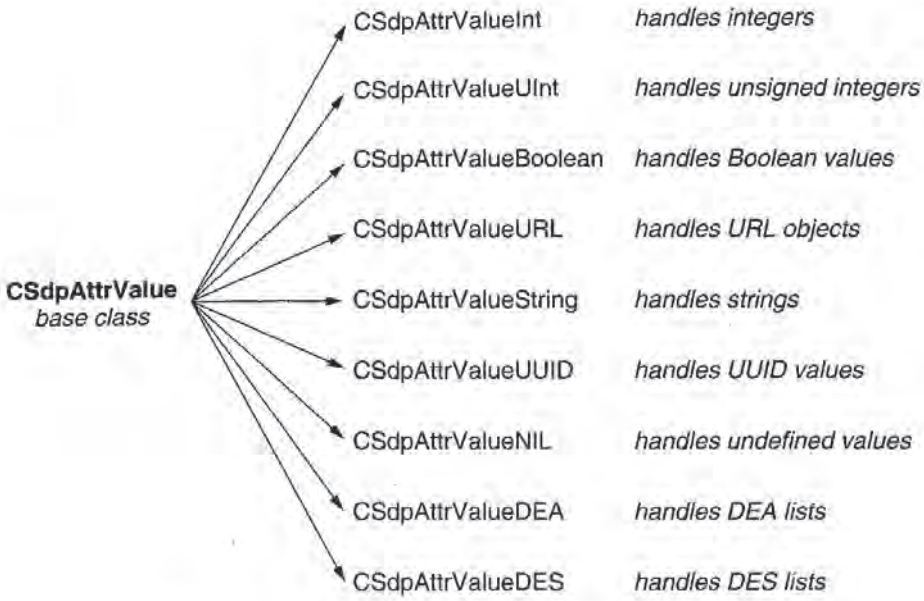

Figure 13.3 Classes that characterize attribute value types.

```
class CSdpAttrValueInt : public CSdpAttrValue
  {
public:
  IMPORT_C static CSdpAttrValueInt* NewIntL(
     const TDesC8 &aInt);
  virtual ~CSdpAttrValueInt();
  virtual TSdpElementType Type() const;
  virtual TUint DataSize() const;
  virtual TInt Int() const;
  virtual TBool DoesIntFit() const;
  virtual const TPtrC8 Des() const;
...
};
```

This class allows the manipulation of integer attribute values in the following way:

- New values are created with the `NewIntL()` method. Note that this method takes a general undefined quantity as a byte buffer.
- The type of the data element is obtained by the `Type()` method. This method returns a value of type `TSdpElementType`, which is simply an enumerated set of nine values (integers) where each represents an attribute type.
- The size of the element is retrieved with the `DataSize()` method.
- The integer value itself is retrieved by the `Int()` method; this converts the byte buffer to the type described by the class, in

this case integer. This is replaced by `UUID()`, for example, for the `CSdpAttrValueUUID` class.

- The `DoesIntFit()` method returns a Boolean value that indicates if the data type will fit into an integer word size.
- The `Des()` method returns the value as a general byte buffer.

Creating an integer attribute is simple; it is executed with one call:

```
CsdpAttrValueInt *intattribute;

intattribute = CSdpAttrValueInt::NewIntL(50);
```

In this example, we have created an attribute whose integer value is `50`.

Attribute lists Attribute lists are a bit more complicated. They must be built, either from simple types or from other lists. The interface, quite frankly, is a little odd; however, when it is written in a certain way, it makes sense.

Both of the data element list classes implement the `MSdpElementBuilder` interface. This interface uses a series of return values of calls to construct a list. Rather than walk through class definitions, let's look at an example. Suppose we wanted to build a DES of integers. We might do this in the following way:

```
CSdpAttrValueDES* integerDES =
   CSdpAttrValueDES::NewDESL(NULL);
integerDES
    ->StartListL()
        ->BuildIntL(TServAttrInt<TInt8>(23))
        ->BuildIntL(TServAttrInt<TInt8>(1))
    ->EndListL();
```

First of all, note that this is actually only two C++ statements. The first is a declaration; the second is a series of function calls that builds a list. The indentation and formatting makes the statement easier to read and understand. Second, this code builds a DES made up of two integers: 23 and 1. The result is one attribute—held in the variable `integerDES`—whose value is a list.

Lists are built following the pattern of the above code. A list is started with a call to `StartListL()` and completed with a call to `EndListL()`. To add something to the list, it must be enclosed in a call to a `Build`*`type`*`L()` method, where the *`type`* is set up to specify an attribute value type. If you are building a nested data

element list, you must also include nested calls to `StartListL()` and `EndListL()`.

For example, let's say we wanted to build an attribute that contained a list of Bluetooth addresses. Each address is a 48-bit integer, so we can encapsulate it in two 32-bit integers. We might create this attribute as follows:

```
CSdpAttrValueDES* addresslist =
CSdpAttrValueDES::NewDESL(NULL);
addresslist
    ->StartListL()
        ->BuildDESL()
            ->StartListL()
              ->BuildIntL(TServAttrInt<TUint32>(TUint32(0x02)))
              ->BuildIntL(TServAttrInt<TUint32>(TUint32(0x817211d)))
            ->EndListL()
        ->BuildDESL()
            ->StartListL()
              ->BuildIntL(TServAttrInt<TUint32>(TUint32(0x0f)))
              ->BuildIntL(TServAttrInt<TUint32>(TUint32(0x620300f)))
            ->EndListL()
    ->EndListL();
```

This code builds an attribute whose value is a list with two nested lists. Each nested list is the two 32-bit integers that together comprise a Bluetooth address.

Working with service records

Service records are identified by their *handle*—a 32-bit integer defined as the class `TSdpServRecordHandle`. These handles are used as keys into the service database. Records are created and deleted by the `CreateServiceRecordL()` and `DeleteRecordL()` methods from the `RSdpDatabase` class. They are created with a UUID or a list of UUIDs that identify them in the database. Once created, the attributes of a record can be updated and modified by using the `UpdateAttributeL()` and `UpdateAttributeL()` methods from the `RSdpDatabase` class.

As an example, let's say we wanted to create an image service record with four attributes—the ones given in Figure 13.2. We could use the code below:

```
#define IMAGE_SERVICE_ID 0x10ff

TSdpServRecordHandle recordHandle = 0;
TUUID image_service_UUID(IMAGE_SERVICE_ID);
database.CreateServiceRecordL(image_service_UUID,
   recordHandle);
```

```
// Now build the attributes
_LIT8(KJPEG,"JPEG");
CSdpAttrValueString* image_format_attribute =
   CSdpAttrValueString::NewStringL(KJPEG);
CSdpAttrValueInt* image_width_attribute =
   CSdpAttrValueInt::NewIntL(320);
CSdpAttrValueInt* image_height_attribute =
   CSdpAttrValueInt::NewIntL(240);
CSdpAttrValueInt* pixel_depth_attribute =
   CSdpAttrValueInt::NewIntL(8);

#define IMAGE_FORMAT_ID 0x10f0
#define IMAGE_WIDTH_ID 0x10f1
#define IMAGE_HEIGHT_ID 0x10f2
#define PIXEL_DEPTH_ID 0x10f3

// Apply the attributes to the records
database.UpdateAttributeL(recordHandle, IMAGE_FORMAT_ID,
   *image_format_attribute);
database.UpdateAttributeL(recordHandle, IMAGE_WIDTH_ID,
   *image_width_attribute);
database.UpdateAttributeL(recordHandle, IMAGE_HEIGHT_ID,
   *image_height_attribute);
database.UpdateAttributeL(recordHandle, PIXEL_WIDTH_ID,
   *pixel_width_attribute);
```

Note a few things about this example. First, when a service record is created in the database, the record handle given as a parameter is filled in with a real value. This means that the value given to the record handle in the code—the value 0—is a "blank" value, a placeholder. Second, let me reiterate that I have arbitrarily assigned ID numbers; these are typically preassigned by the Bluetooth Consortium.

13.2.5 Some Perspective

At this point, you should catch your breath and review the last several pages. Some perspective on device and service discovery is in order.

It should now be quite clear that these discovery methods add a significant new layer of complexity to Bluetooth communications. While the models used define a flexible and adaptive system for all sorts of devices and services, searching for devices and probing service databases can be complex and time-consuming. From the searching device's point-of-view, we have a four-stage process: setup the query, make the query, wait for the query to return, process the query's results. This four-stage process is used for device discovery, for service discovery, and for each attribute that

we need to process. On a memory and CPU limited device, this can amount to a big problem.

Consider an example. Suppose you and your Bluetooth-enabled computer enter a laboratory with a need to take a picture from a digital camera—one you have with you—and to print it on a printer. You know that you have a Bluetooth-enabled camera, but you do not know if a Bluetooth printer is in the room. There are five other Bluetooth devices in the room, however. While your computer can connect directly to the camera—you already have the address preconfigured in the application you are using—you must attempt to discover a printer. So, when your computer commences device discovery, it will have to examine six device replies (the camera will probably reply as well) to find printer attributes. If it finds printer attributes in a service, your computer will have to probe further to determine if it can use the printer. Hopefully, the right printer is one of the first to reply to the device discovery probe and not the last one!

While discovery does indeed appear to be a lot of work, note that we do not have much of a choice here. In a general Bluetooth environment, devices move in and out of a network's range regularly. Applications provide increasingly customized services, and devices can make few assumptions.

So programmers might have to be innovative and sparing as they implement generalized applications. It is also helpful if we can strike some kind of compromise between full device and attribute discovery and using a complete set of assumptions. That compromise can be found in the concept of class settings.

13.2.6 Class Settings: Some Middle Ground

There is, in fact, a middle ground that is helpful as we try to minimize discovery. The Bluetooth specification defines two types of coarse, descriptive characterizations for a device: the *major* and *minor class settings*. These settings are actually returned to the device with the `TInquirySockAddr` object at device discovery.

You can use these class settings as early indicators of the type of the remote device. Using these settings, you might be able to tell the difference between a printer and digital camera. You still need to check the printer for specific requirements, but you can stop certain types of discovery by using these coarse class settings.

You can use methods from the `TInquirySockAddr` class to examine these class settings:

```
inline TUint16 MajorServiceClass() const;
inline void SetMajorServiceClass(TUint16 aClass);
```

```
inline TUint8 MajorClassOfDevice() const;
inline void SetMajorClassOfDevice(
   TUint8 aMajorClassOfDevice);
inline TUint8 MinorClassOfDevice() const;
inline void SetMinorClassOfDevice(
   TUint8 aMinorClassOfDevice);
```

The class settings are simply integers—codes predefined by the Bluetooth consortium. Notice that you can set your class settings as well. Also notice from the above methods that class settings apply to device characterization as well as to service characterization.

13.3 Communicating over Bluetooth

Now that we know how to get device addresses and service descriptions, we can examine how the Symbian OS supports Bluetooth communication. Here we will revel in the modularity of the Symbian design, because communicating over Bluetooth is based on the now familiar model of socket communication. We will tackle this using our five-stage pattern for covering sockets.

13.3.1 Initializing the System for Bluetooth

To initialize the Symbian OS for Bluetooth communications, we need to connect to the socket server and make it load the Bluetooth drivers in which we are interested. This looks very much like the sequence we used to start up IrDA socket communications. The code fragment below illustrates this.

```
// Start comm server if necessary
result = StartC32();
if (result != KErrNone && result != KErrAlreadyExists)
   User::Leave(result);

// Connect to the socket server
result = socksvr.Connect();
if (result != KErrNone) User::Leave(result);

// Load the protocol
result = socksvr.FindProtocol(_L("BtLinkManager"),
   protocolInfo);
User::LeaveIfError(result);
```

Here, we start the socket server, connect to it, and look for the LMP driver. Loading this driver will force the loading of the other protocol modules as necessary.

If we are to enable remote devices to discover the local device's features, we must also initialize the system for discovery as we discussed in detail previously in this chapter. If the application is to do service database manipulation, some special care must be taken to keep the database server up and listening. When the last connection to the service database is closed, the server shuts down even if an SDP connection is open. This means that an application must take care to keep connections (database subsessions) open until all database manipulation is completed.

It is useful for a Bluetooth device to have a name. We can get and set the name of a device through the host resolver. The definitions below show the methods that are used to do this:

```
TInt GetHostName(TDes &aName);
void GetHostName(TDes &aName,TRequestStatus &aStatus);
TInt SetHostName(const TDesC &aName);
```

We have seen these before with the `RHostResolver` definition. The `GetHostName()` method will retrieve the name of the local device, placing it in the `aName` parameter. The `SetHostName()` method will set the name on the local device.

13.3.2 Opening and Configuring a Bluetooth Socket

Opening and configuring a Bluetooth socket looks almost like using any other socket. Assuming we have the information we need—a device address and a service identifier—opening and configuring a socket is quite standard.

Consider the code below from the `TodoX` example.

```
TBool TodoXferBt::OpenL()
{
  TInt result;

  result = sock.Open(socksvr, KBTAddrFamily,
                     KSockSeqPacket, KL2CAP);
  User::LeaveIfError(result);

return true;
}
```

The socket used here is again from the `RSocket` class. Notice that we have used `KBTAddrFamily` for the address family. The socket type specification changes based on the protocol that is being used. We must use `KSockStream` if we use RFCOMM; we use `KSockSeqPacket` if we use L2CAP.

We can configure a socket at three different levels using `Ioctl()` calls:

- Low-level configuration, i.e., levels of protocol below L2CAP, can be configured through Host Controller Interface (HCI) commands to the `Ioctl()` method for a socket.
- There are also `Ioctl()` calls that will configure settings for the L2CAP protocol level.
- Settings for RFCOMM sockets can also be done through `Ioctl()`. RFCOMM sockets also have options that can be retrieved through the `GetOpt()` method of the `RSocket` class.

13.3.3 Connecting Client Sockets

Clients connect to servers using the `RSocket::Connect()` method. As we have seen before, this method needs a valid address and service specification to which to connect the socket. The service specification is set in the address object. This object can take one of several forms, all of which are derived through the inheritance tree from the `TSockAddr` class:

- It can be a `TBTSockAddr` object. This class has a `SetBTAddr()` method to set the Bluetooth device address directly, and a `SetPort()` method (from `TSockAddr`) to set the service specifier.
- It can be a `TInquirySockAddr` object, since this is the class of the information that results from device discovery. There is no need to set the device address but the service ID must be set with the `SetPort()` method.
- It can be a protocol-specific address, defined as either a `TL2CAPSockAddr` object or a `TRfcommSockAddr` object, specifying L2CAP or RFCOMM protocols, respectively. This class borrows methods to set the device address and service specifiers from the `TBTSockAddr` class.

We should make an observation here about service IDs. The L2CAP layer is meant to implement multiple services multiplexed and identified by service specifiers. RFCOMM is meant to implement serial communication behavior. Serial communication does not need multiple service specification; therefore, RFCOMM uses only one service ID, whose integer value is `3`.

Once our socket is created, the call to `Connect()` looks the same as with other protocols. The code below is from the `TodoX` example:

```
addr.SetPort(15);
sock.Connect(addr,status);

iSendingState = ESC;
SetActive();
```

The call to `Connect()` is asynchronous, and `TodoX` uses an active object for the Bluetooth exchange. Therefore, we also set up the connection state, and we call `SetActive()`. Notice here that we have specifically set up the service specifier value as `15`, chosen arbitrarily.

13.3.4 Binding Server Sockets

Message servers (i.e., receivers) go through the usual procedure in setting up a listener for Bluetooth connections: bind an opened socket to a local address, set up a listening queue on that socket, and open a "blank" socket to connect to an incoming connection.

For Bluetooth servers, we must bind the listening socket to a specific service ID. We do not need to specifically set the Bluetooth device address, which means the server will connect to any device, but we must set the address and listen only for connections on a specific port.

The code below from the `TodoX` example has a familiar pattern:

```
void TodoXferBt::ConfigureReceiver()
{
   TInt result;

   // ... configure security ...

   result = iListener.Open(socksvr, KBTAddrFamily,
      KSockSeqPacket, KL2CAP);
   User::LeaveIfError(result);
   addr.SetPort(15);
   result = iListener.Bind(addr);
   User::LeaveIfError(result);
   result = iListener.Listen(5);
   User::LeaveIfError(result);
}
```

This example sets up a listener socket for any device on port `15`, with a listener queue size of 5.

To receive a connection, the service must call the listening socket's `Accept()` method with a "blank" socket. When `Accept()` returns, the socket will be defined and the two endpoints will have a data channel over which they can communicate.

13.3.5 Exchanging Data

As two devices are poised to send data between them, each side has a connected socket to use for the data transmission. At this point, the standard API for socket communication is used. The protocol issues—including initiation and termination—are still valid and are not specialized because we are using Bluetooth.

13.3.6 Closing a Socket

Bluetooth sockets are closed in the normal manner, by draining the socket of pending I/O requests, shutting down the socket, and then shutting down the socket server.

13.4 The Bluetooth Security Manager

One of the benefits of designing the Bluetooth protocol suite after other protocols had been designed and tested is that its designers could include features and details that the other protocol suites lacked. One of these details is security. Bluetooth pays attention to security issues in situations where other protocols do not. Bluetooth protocols negotiate certain security settings when incoming connections are made to devices. Specifically, two connecting devices will negotiate whether authentication, authorization, and encryption are required.

The Symbian OS implements this negotiation through the use of a *security manager*. The security manager is a server that protects and manages the resources associated with incoming connections. The `RBTMan` class encapsulates the functionality of the security manager. This manager maintains a database of security settings, characterized in the Symbian OS by the `RBTSecuritySettings` class. A specific collection of settings is handled by the `TBTServiceSecurity` class. To control the security manager, an application must first connect to it, open the security database through an `RBTSecuritySettings` object, and then set up a policy on a connection over a specific service specifier by registering a `TBTServiceSecurity` object.

The `TBTServiceSecurity` class has the following definition:

```
class TBTServiceSecurity
{
public:
  IMPORT_C TBTServiceSecurity(TUid aUid,TInt aProtocolID,
     TInt aChannelID);
  IMPORT_C TBTServiceSecurity(
     const TBTServiceSecurity &aService);
```

```
  IMPORT_C TBTServiceSecurity();
  IMPORT_C void SetUid(TUid aUid);
  IMPORT_C void SetProtocolID(TInt aProtocolID);
  IMPORT_C void SetChannelID(TInt aChannelID);
  IMPORT_C void SetAuthentication(TBool aPreference);
  IMPORT_C void SetAuthorisation(TBool aPreference);
  IMPORT_C void SetEncryption(TBool aPreference);
  IMPORT_C void SetDenied(TBool aPreference);
  IMPORT_C TBool AuthorisationRequired() const;
  IMPORT_C TBool EncryptionRequired() const;
  IMPORT_C TBool AuthenticationRequired() const;
  IMPORT_C TBool Denied() const;
  IMPORT_C TUid Uid() const;
  IMPORT_C TInt ProtocolID() const;
  IMPORT_C TInt ChannelID() const;
private:
...
};
```

A set of service settings is identified by a UID and the service specifier and protocol over which it is defined. Using the above class interface, settings can be turned on or off for authentication, authorization, and encryption. We can even flat out deny a specific protocol connection through these settings. There are also methods to determine the security settings for a particular connection.

We might establish a set of security specifications as in the following example:

```
TBTServiceSecurity securityspecs;
TUid specsUID;
specsUID.iUid = 0x01ff; //chosen arbitrarily

securityspecs.SetUid(specsUID);
securityspecs.SetChannelID(15);
securityspecs.SetProtocolID(KSolBtL2CAP);

securityspecs.SetAuthentication(EFalse);
securityspecs.SetAuthorisation(EFalse);
securityspecs.SetEncryption(ETrue);
```

The code above sets up a security specification for connections over L2CAP using a service ID of `15`. The specification stipulates no authentication or authorization is necessary, but encryption must be used.

The `RBTSecuritySettings` class has the following definition:

```
class RBTSecuritySettings : public RBTManSubSession
  {
public:
  IMPORT_C TInt Open(RBTMan& aSession);
  IMPORT_C void Close();
```

```
  IMPORT_C void RegisterService(
     const TBTServiceSecurity &aService,
     TRequestStatus &aStatus);
  IMPORT_C void UnregisterService(
     const TBTServiceSecurity &aService,
     TRequestStatus &aStatus);
    ...
};
```

Using this class, an application can open and close the security database and register and unregister a set of security specifications.

The RBTMan class has a rather simple definition:

```
class RBTMan : public RSessionBase
{
public:
  IMPORT_C RBTMan();
  IMPORT_C TInt Connect();
  IMPORT_C TVersion Version() const;
};
```

An application can use this class to connect to the server and tell its version. Via an inherited method from the RSessionBase class, a security manager session is closed with a Close() method call.

Given that we have the security specifications we used in previous examples, we might enter them into the security database with the code below:

```
RBTMan security_manager;
RBTSecuritySettings security_database;
TBTServiceSecurity securityspecs;
TRequestStatus status;
TInt result;

// ... define the security specifications for L2CAP/service 15

result = security_manager.Connect();
User::LeaveIfError(result);
Result = security_database.Open(security_manager));
User::LeaveIfError(result);

security_database.RegisterService(securityspecs, status);
User::WaitForRequest(status);
```

Notice that the RegisterService() method is an asynchronous call that requires us to wait for a result. The UnRegisterService() method is also asynchronous. Most security settings are imposed by the server—the recipient of a connection. The settings are negotiated as the connection is being made, and the client will automatically adjust its settings to match those demanded by the server. There is a way for the client to request settings: it must call

the `Ioctl()` method for the socket being used as the connection. An example is below:

```
sock.Ioctl(KHCIAuthRequestIoctl, status, NULL, KSolBtHCI);
User::WaitForRequest(status);
```

This sets the outgoing socket connection to specifically request that authentication be used. Note that, in a settings negotiation, either side may reject the settings proposed by the negotiating partner.

13.5 An Example: Bluetooth Instant Messaging

To complete this chapter, I would like to work through one more example. Suppose we wanted to implement an application that would send short text messages between two Bluetooth devices—a kind of Bluetooth instant messenger. Let's design and construct an application to do this and see what it entails.

13.5.1 Designing the Application

The Bluetooth Instant Messenger (BTIM) application is simple in concept and composed of three parts:

- *The user interface:* Obviously, any application for a Symbian OS device must have a user interface. While it is not really the concern of this chapter, suffice it to say that we must have a simple and effective interface. An example interface is shown in Figure 13.4.
- *The sender:* This is an active object that engages a receiver in a protocol to send a message over Bluetooth. This message sending "protocol" is simply a text string that is acknowledged by the receiver. The state diagram for this active object is in Figure 13.5.
- *The receiver:* This is an active object that acts as a kind of display server. It fields connections from senders, engages in a message protocol over Bluetooth, and writes the received message to the receiving device's display. The state diagram for the server's protocol is in Figure 13.6.

Note that there are two active objects at work concurrently in this application. The message sender's client works with a remote receiver's server to get a message to the remote device's display. However, each device in the BTIM application can be both a sender and a receiver, hence the need for two active objects.

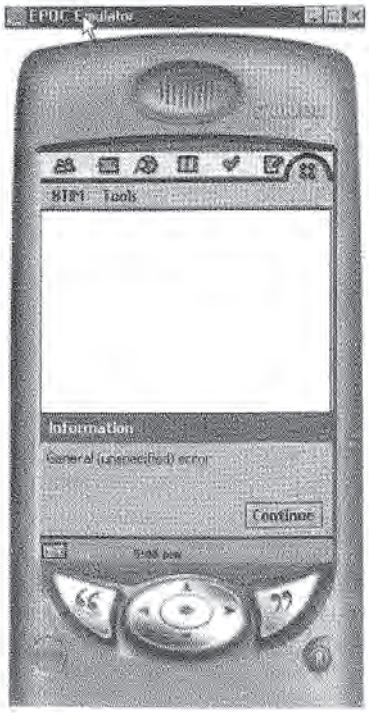

Figure 13.4 The BTIM user interface.

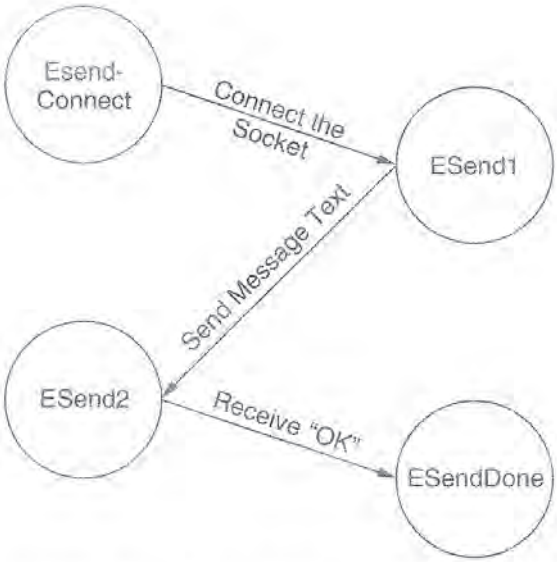

Figure 13.5 State diagram for the sender's active object.

Consider the act of sending a message. Each message must have a destination attached. However, as is usually the case with Bluetooth, it is not sufficient to set the receiver device once and have this receiver be used for each message we send. Instead, we must set the receiver for each message that goes out. This means device discovery must be performed for each message that is sent.

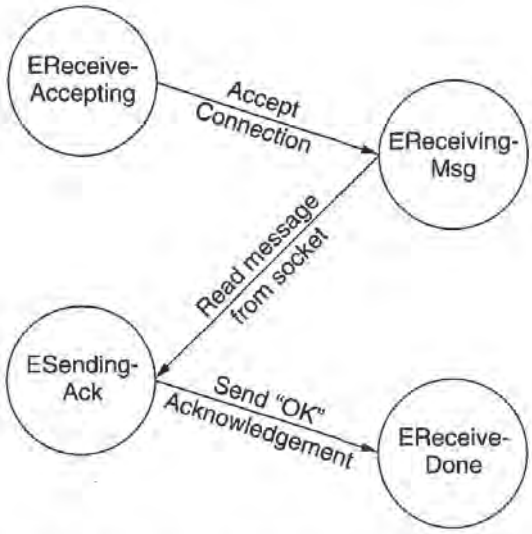

Figure 13.6 State diagram for the receiver's active object.

Consider how BTIM would receive messages. Receiving means an infinite iteration of accepting a socket connection, taking a text message over this socket, and writing this text to the display. The receiver will have to pull the name of the sender's device from the socket connection. The receiver will also have to choose the options to negotiate for this connection; in the implementation, I have chosen the easy way: no authentication, no authorization, and no encryption is used.

13.5.2 Initializing the Application

The application is quite simple. The interface is a container with a single textbox (an `CEikTextListBox` object) inside. The receiver object is created at the same time the interface is constructed and initialized. The code below shows how this is done:

```
void CBTIMAppUi::ConstructL()
{
   // Allow base class (CEikAppUi) to perform necessary
   // construction
   CQikAppUi::ConstructL();

   // Build the container
   iContainer = CBTIMContainer::NewL();

   // The container is added to the control stack (for key
   // event handling).
   AddToStackL(iContainer);

   // Start the receiver
   iReceiver = BTIMReceiver::NewL(this);
```

```
    iReceiver->Init();
    iReceiver->OpenL();
    iReceiver->Configure();
    iReceiver->StartReceive();
}
```

So after the interface has been built, we instantiate and initialize the receiver object. The call to the `StartReceive()` method initiates the parallel execution of the active object thread.

The interface destructor is below:

```
CBTIMAppUi::~CBTIMAppUi()
{
    iReceiver->Close();
    delete iReceiver;
    RemoveFromStack(iContainer);
    delete iContainer;
}
```

The destructor of the `CBTIMAppUi` object destroys the receiver object when it destroys the user interface container.

13.5.3 The Sender Object

The implementation of the sender object takes the form of the `BTIMSender` class and follows our five-stage definition for sockets. In the BTIM application, sending a message happens in response to the user tapping a button on the application toolbar. In response to this button tap, we call the `SendMessage()` method, whose code is below:

```
void CBTIMAppUi::SendMessage()
{
    TBuf<40> msg;
    CEikDialog* dialog =
        new (ELeave)
CBTIMMessageTextDialog(&msg);
    if (dialog->ExecuteLD(R_MESSAGE_TEXT_DIALOG)) {
        iSender = BTIMSender::NewL(this);
        iSender->Init();
        iSender->OpenL();
        iSender->Configure();
        iSender->SendMsg(msg);
        iSender->Close();
    }
}
```

We pop up a dialog to capture the message text, and then go through the five stages of socket usage.

Initializing the socket

We initialize a socket by starting and connecting to the socket server, loading the proper protocol, and opening a host resolver socket for device discovery. The `Init()` method is below:

```
TInt BTIMSender::Init()
{
    TInt result;
    TUint nProtocols;
    TNameEntry nentry;
    THostName hostname;

  // Start comm server if necessary
  result = StartC32();
  if (result != KErrNone && result != KErrAlreadyExists)
      User::Leave(result);

  // Connect to the socket server
  result = socksvr.Connect();
  if (result != KErrNone) User::Leave(result);

  // Load the protocol
  result = socksvr.FindProtocol(_L("BTLinkManager"),
     protocolInfo);
  User::LeaveIfError(result);

  // get the resolver ready
  result = resolver.Open(socksvr,
     protocolInfo.iAddrFamily, protocolInfo.iProtocol);
  User::LeaveIfError(result);

  return ETrue;
}
```

Opening the Bluetooth connection

For the sender, the `OpenL()` method simply involves opening the socket that we will use to send data to a receiver. The `OpenL()` method is below:

```
TBool BTIMSender::OpenL()
{
   TInt result;

   result = sock.Open(socksvr, KBTAddrFamily,
                      KSockSeqPacket, KL2CAP);
   User::LeaveIfError(result);

   return ETrue;
}
```

Note that we are using the L2CAP protocol, so I use the `KBTAddrFamily` address type and a socket type of `KSockSeqPacket`.

Configuration and device discovery

There really is no need for further configuration for a Bluetooth socket beyond the parameters given when the socket is opened, so I chose to do device discovery at configuration time. The `Configure()` method code is below:

```
void BTIMSender::Configure()
{
   TInt result;
   TNameEntry tnentry;

   // discover the device
   addr.SetIAC(KGIAC);
   addr.SetAction(KHostResInquiry);
   resolver.GetByAddress(addr, tnentry, status);
   User::WaitForRequest(status);

   // discover the service we want
   if (status == KErrNone) {
      tnentry.setName(SelectionUI(tnentry));
   } else {
      iAppUi->ShowErrMessage(_L("Cannot discover any other"),
                             _L("Bluetooth devices."));
      User::Leave(status.Int());
   }
}
```

This is fairly standard device discovery, using by-address discovery and setting up an unlimited query. The response comes back in the `tnentry` variable.

If there is a single device that needs to be discovered, we can pick the first response and use that one as the destination to which to send the message. However, if multiple devices have responded, the user must pick one. This is a common task, and the Symbian OS has provided a device selection user interface to perform it.

This device selection user interface is displayed by using an object from the `RNotifier` class and some selection filter classes. Consider the example code below:

```
TInt result;
TRequestStatus status;
RNotifier notifier;
TBTDeviceSelectionParams selectionFilter;
TUUID targetServiceClass(0x2345); // arbitrary, example ID
TBTDeviceResponseParams response;
TPtrC name;
```

```
// Connect to the notifier server
result = notifier.Connect();
User::LeaveIfError(result);

// Set up the parameters
selectionFilter.SetUUID(targetServiceClass);
TBTDeviceSelectionParamsPckg pckg(selectionFilter);
TBTDeviceResponseParamsPckg responsePckg(response);

// Start up the notifier
notifier.StartNotifierAndGetResponse(status,
   KDeviceSelectionNotifierUid, pckg, responsePckg);
User::WaitForRequest(status);

// Handle the notification
if (status.Int() == KErrNone) {
    if (resultPckg.IsValidDeviceName()) {
       name.Set(resultPckg().DeviceName());
    }
}

// Shut down the notifier
notifier.CancelNotifier(KDeviceSelectionNotifierUid);
notifier.Close();
```

Notice that we set up all the data, then call the `StartNotifierAndGetResponse()` method for the `RNotifier` object. By sending it the `KDeviceSelectionNotifierUid`, we get the proper dialog. Once the user has selected a device name, the above code extracts it to the name variable.

Sending the message

We engage in the message exchange protocol when the `SendMsg()` method is called with a text message as the parameter. Its code is below:

```
void BTIMSender::SendMsg(TDesC &aMsg)
{
   buffer.Copy(aMsg);

   addr.SetPort(BTIM_SENDING_PORT);
   sock.Connect(addr,status);
   iConnectionState = ESendConnect;
   SetActive();
}
```

In this code, we copy the message to an 8-bit buffer, set the address sending specifier to a specific value (thereby avoiding the hassle of service discovery), then try to connect the socket to the remote device.

The `RunL()` method implements the message exchange protocol. Its code is as follows:

```
void BTIMSender::RunL()
{
   TInt pos,err,i,result;

   switch (iConnectionState) {
      case ESendConnect:
         if (status == KErrNone) {
            iAppUi->SetProgress(_L("Sending Message"));
            Send(buffer);
            iConnectionState = ESend1;
         } else {
            // handle the error
         }
         break;

      case ESend1:
         if (status == KErrNone) {
            iAppUi->SetProgress
              (_L("Awaiting acknowledgement"));
            Receive(buffer);
            iConnectionState = ESend2;
         } else {
            // handle the error
         }
         break;

      case ESend2:
         if (status == KErrNone) {
            if (buffer != _L8("OK")) {
               iConnectionState = ESendError;
            } else {
              iConnectionState = ESendDone;
            }
         } else {
            // handle the error
         }
         break;
   }

   if (iConnectionState == ESendDone) {
     socksvr.Close();
     sock.Close();
     delete this;
   } else if (iConnectionState == ESendError) {
     // handle the error
   }
}
```

This code uses the same `Send()` and `Receive()` methods as the `TodoX` example; the format of each message includes the length of

the message as the first byte. Notice that when the object is done sending the message, it deletes itself. This is so the BTIM application object can initiate a message send operation and then leave the sending work to the active object.

13.5.4 The Receiver Object

The implementation of the receiver is via the `BTIMReceiver` class and follows the five-stage implementation pattern to which we are accustomed. The sequence of starting and terminating a receiver was already given in the code for the interface construction.

Initializing the receiver connection

Initialization of the receiver is much like that for the sender, with the exception that we do not need a host resolver. Receivers do not do device discovery. Therefore, like the sender, we start the communication server, connect to the socket server, and load the right protocol module. This code is below:

```
TInt BTIMReceiver::Init()
{
    TInt result;
    TUint nProtocols;
    TNameEntry nentry;
    THostName hostname;

  // Start comm server if necessary
  result = StartC32();
  if (result != KErrNone && result != KErrAlreadyExists)
          User::Leave(result);

  // Connect to the socket server
  result = socksvr.Connect();
  if (result != KErrNone) User::Leave(result);

  // Load the protocol
  result = socksvr.FindProtocol(_L("L2CAP"),
     protocolInfo);
  User::LeaveIfError(result);
  return ETrue;
}
```

Notice that we load the L2CAP protocol module instead of the LMP module. We need the LMP module for device discovery but not for socket connections.

Modularity vs Efficiency

For the purposes of description in this book, I have purposefully structured all socket code the same. This is a very modular way to look at socket usage. However, it is not the most efficient. In the BTIM application, it is better to pull common elements out of both objects and combine them in the main application.

An interesting question is this: how much time does this save? For instance, while an application can certainly load protocol modules twice without any errors, it does waste time to do this. But how much? Is the modularity of the design I have worth the time used?

Perform an experiment to find out. Make a guess as to whether significant time is saved, make some adjustments to the software, and then run both versions. Make timing measurements and see if your original guess was correct.

Opening the Bluetooth connection

The `OpenL()` method opens both the listener socket and the first blank socket for communicating with the remote device. The code is below:

```
TBool BTIMReceiver::OpenL()
{
   TInt result;

   result = iListener.Open(socksvr, KBTAddrFamily,
                 KSockSeqPacket, KL2CAP);
   User::LeaveIfError(result);
   result = iServiceSocket.Open(socksvr, KBTAddrFamily,
                  KSockSeqPacket, KL2CAP);
   User::LeaveIfError(result);

   return true;
}
```

Configuring the receiver socket

Configuration of the receiver socket must do two things: it must configure the listening socket, and it must set up the security parameters for incoming connections. The code to do this is below:

```
void BTIMReceiver::Configure()
{
    TInt result;
    RBTMan manager;
    RBTSecuritySettings secdatabase;
    TBTServiceSecurity settings;
    TUid settingsUID;

    // Set up the listening socket
    addr.SetPort(BTIM_SENDING_PORT);
    result = iListener.Bind(addr);
    User::LeaveIfError(result);
    result = iListener.Listen(5);
    User::LeaveIfError(result);

    // Connect and open the database
    result = manager.Connect();
    User::LeaveIfError(result);
    result = secdatabase.Open(manager);
    User::LeaveIfError(result);

    // Set up the settings
    settingsUID.iUid = 0x01ff;
    settings.SetUid(settingsUID);
    settings.SetChannelID(BTIM_SENDING_PORT);
    settings.SetProtocolID(KSolBtL2CAP);
    settings.SetAuthentication(EFalse);
    settings.SetAuthorisation(EFalse);
    settings.SetEncryption(EFalse);

    // Register the settings
    secdatabase.RegisterService(settings, status);
    User::WaitForRequest(status);
}
```

As we stated above, the security settings do not use authentication, authorization, or encryption.

Receiving a message

The receiver object starts receiving when the `StartReceiver()` method is called. This method simply makes an `Accept()` call and initiates the active object thread execution. The code is as follows:

```
void BTIMReceiver::StartReceive()
{
    TInt result;

    iListener.Accept(iServiceSocket, status);

    iConnectionState = EReceiveAccepting;
    SetActive();
}
```

The `RunL()` method implements the message receipt protocol.

```
void BTIMReceiver::RunL()
{
   TInt pos,err,i,result;

   switch (iConnectionState) {
      case EReceiveAccepting:
         if (status == KErrNone) {
            iAppUi->SetProgress(_L("Establishing protocol"));
            Receive(buffer);
            iConnectionState = EReceivingMsg;
         } else {
            // handle the error
         }
         break;

      case EReceivingMsg:
         if (status == KErrNone) {
            iAppUi->SetProgress
              (_L("Awaiting acknowledgement"));
            buffer.Copy(_L8("OK"));
            Send(buffer);
            iConnectionState = ESendingAck;
         } else {
            // handle the error
         }
         break;

      case ESendingAck:
         if (status == KErrNone) {
            if (buffer != _L8("OK")) {
               iConnectionState = EReceiveError;
            } else {
              iConnectionState = EReceiveDone;
            }
         } else {
            // handle the error
         }
         break;
   }

   if (iConnectionState == EReceiveDone) {
       iServiceSocket.Close();
       StartReceive();
   } else if (iConnectionState == EReceiveError) {
      // handle the error
   }

}
```

The receiver uses `Send()` and `Receive()` methods as the sender does to send and receive strings. Note that when a message receipt is done, an application method is called to display the text, the service socket is closed, and the process is started again through a

call to `StartReceive()`. This is why the service socket is opened in that method rather than in the `OpenL()` method.

13.6 Summary

In this chapter, we discussed the support that the Symbian OS gives to Bluetooth protocols and how we are to use the APIs that result. This has been a long chapter, with much space devoted to a key element of Bluetooth technology: device and service discovery. The Symbian OS supports the rest of the Bluetooth protocol suite through extensions to the socket model. Socket support looks very much like the support for other protocol suites. We also spent time on a large example: Bluetooth instant messaging.

In the next chapter, we take on a bridge of sorts: telephony. It is not really a transport technology nor a content technology. However, it addresses a distinctive feature of Symbian OS devices, one that demonstrates the convergence of computing and communication.

14

Telephony

The convergence of telephony and handheld computing devices has had a long history. As soon as the first devices sported a serial port, telephony services could be accessed by connecting a modem and sending it commands to dial or answer phone calls. As integrated devices were developed, control over telephony functions became a more crucial issue for Symbian designers. Telephony functionality was inextricably bound to Symbian OS devices with the Ericsson R380 and Nokia 9210/9290. These devices demonstrate how telephony communication and computing capability—integrating data and voice capability—can be mutually beneficial.

This chapter will examine how Symbian OS supports telephony. In a way, this chapter represents a transition from transport technologies to content technologies. It lies between the two; telephony is neither transport nor content, yet can be used for both. The wide variety of telephony functions and uses are provided by the Symbian OS ETel telephony model.

This chapter will review the details of Symbian OS telephony services. To fully appreciate the ETel model, it is pertinent that we start by examining all of the different situations it was designed to address. We will then take an overview look at the structure of the API. Then we will dig into the details of ETel by looking at the four main objects in its design. And finally we will wrap up by looking at faxing as a special case of telephony.

14.1 The Various Forms of Telephony

Communication using a conventional analog telephone basically takes two forms: data and voice communication. Data communication over a telephone is handled by a modem, the digital-to-analog converter we discussed in Chapter 4. Fax messages are image files

converted to digital format; faxes are sent between computers and fax machines using fax modems. The computer controls the modem by using a set of commands. The Hayes command set, for example, is by far the most widely used set of commands for modems. Hayes-format commands take the form of "AT" commands: the character sequence "AT" followed by characters to command the modem to do certain tasks. To initialize a modem and cause it to dial a number, for example, we might give it the following command string:

```
ATQ0E1DT555-3454
```

This string tells the modem to send result codes back to the computer in response to commands, echo back the commands sent to the modem, and tone-dial the number `555-3454`.

Voice communication via telephone is effectively a peer-to-peer connection, where the "address" of the remote device is the telephone number being dialed. The telephone number is accepted by the telephone network switching infrastructure and a stable connection is made between the local and remote devices, until one or both terminate the connection by "hanging up". A telephone can handle several telephone lines; each line can typically multiplex several phone calls at once.

Telephone networks can be wired (so-called "landline" networks) or wireless (often termed "cellular" networks—I prefer the term "mobile"). Of the possible equipment (or handset) and switching network combinations, digital choices have emerged as the most promising for the future. A GSM phone is a good example of digital phone equipment using a digital connection network. As we discussed briefly in Chapter 4, GSM is a packet-based protocol that sends data packets of a fixed size for a fixed time interval. The phone handset must also be a digital device to use the digital network, and some offer the capability of being controlled by a computer. My Nokia 8290 handset, for example, has an infrared port that allows it to accept commands from an external device.

14.2 A Structural Overview

The model Symbian OS uses is abstract enough to provide the application programmer with a consistent, standard interface, no matter what kind of device is being used.

The first thing to point out about this model is that telephony functionality on Symbian OS is viewed as a resource and, as with all

resources on Symbian OS, there is a server to protect and manage it. As a first step to using a telephony device, then, an application needs to connect to the root telephony server.

The remainder of the Symbian telephony model closely follows experience in using telephones. Figure 14.1 shows a diagram of this Symbian view. The model looks at telephony as a collection of *phones*. Each phone is an abstraction of a telephony device (e.g., a modem or a GSM phone). Through this abstraction, we can access a device's status and capabilities and be notified if changes occur to a device's properties. A phone can have one or more *lines*. An application can access the status and capabilities of a line, just as it can for a phone, and can be notified if any of these change.

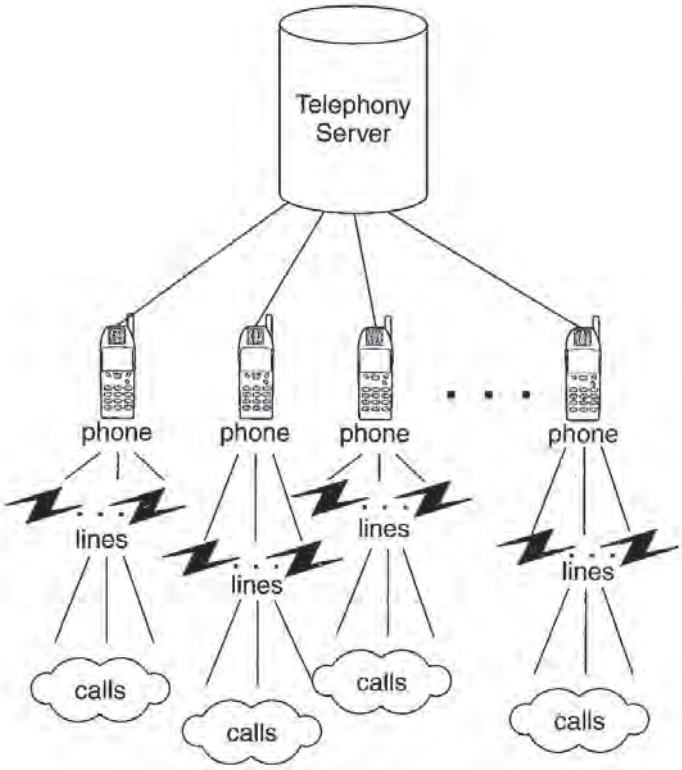

Figure 14.1 The ETel phone model.

The actual use of a line from a phone is designated as a *call*. A line can have zero or more active calls. A call can dial a number, wait for an incoming connection, and be terminated. As with lines, an application can get status and capabilities information for a call and be notified of changes to a call.

These abstractions combine to form the ETel model. The heart of this model's implementation is in the TSY module. By integrating the specific implementation of this model for a particular phone

type into a module, Symbian OS designers ensured that the API for this functionality would remain the same across different phones, and the application programmer is free from worrying about the implementation specifics for a particular phone. In addition, when a new phone must be integrated into this framework, it amounts to a new TSY module implementation.

It is important to note what ETel is *not*. It *is* a phone access model; it *is not* a model for anything beyond phone access. For example, while ETel supports phone dialing, it does not support address translation when dialing—for example, the altering of the address dialed to reflect location information. As another example, ETel supports call connection and termination, but it does not support call management or queuing of calls to be made. ETel is specifically targeted at access to phone features and nothing more.

TSYs Supported by Symbian OS

TSYs are specifically designed to plug into the telephony server and provide access to telephony features. Figure 14.2 shows where TSY modules fit in the ETel infrastructure. Search your Symbian device or emulator and look for TSY modules provided with the device. In version 6.1 of Symbian OS, you should find at least three TSYs that are provided with the distribution.

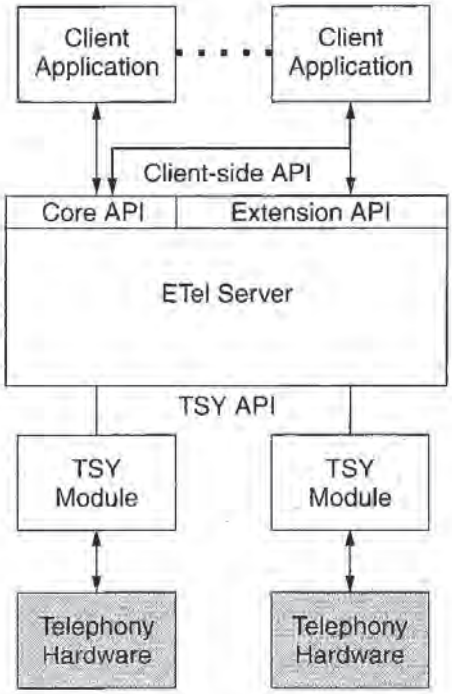

Figure 14.2 ETel structural diagram.

14.3 Components of the ETel Model

The ETel model has four components: the root telephony server, the phone component, the line component, and the call object. These are depicted in Figure 14.1.

14.3.1 The Root Server

The root server manages access to the telephony system. It is implemented by the `RTelServer` class. We will examine the definition of this class in several pieces.

As with all resources managed by servers, before telephony can be used we must connect to the telephony server. We do this with the `Connect()` function, defined below:

```
IMPORT_C TInt Connect(TInt aMessageSlots
   =KDefaultMessageSlots);
```

Using this function, we connect to the telephony server and specify how many message slots we need. Recall from Chapter 8 that a single message slot is a communication channel in one direction. The default number of slots assigned is 32. The `RTelServer` class is a subclass of the `RSessionBase` class, and therefore inherits the `Close()` function, which is used to shut down an active telephony server session.

Once a connection is established, the TSY module that is needed should be loaded. TSY modules can be manipulated through the functions defined below:

```
IMPORT_C TInt LoadPhoneModule(
   const TDesC &aFileName) const;
IMPORT_C TInt UnloadPhoneModule(
   const TDesC &aFileName) const;
```

The first function loads a TSY module, and the second function removes a TSY module. The TSY modules are specified by name via a string to the parameter of each function. The name should be specified without the `.tsy` suffix; the system will supply this as necessary. For example, we could load the GSM TSY module as `gsmbsc.tsy` or `gsmbsc`; either form will work.

Once the appropriate TSY module has been loaded, we can make queries about its properties. These queries take the form of telephony server functions, defined below:

```
inline TVersion Version() const;
IMPORT_C TInt EnumeratePhones(TInt &aNoOfPhones) const;
IMPORT_C TInt GetPhoneInfo(const TInt aIndex,
   TPhoneInfo &aInfo) const;
```

The first function retrieves the version number of the TSY module and returns that version as either a build number or a combination of major and minor version numbers. The `EnumeratePhones()` function will return in its parameter the total number of phones supported by TSY modules loaded by the telephony server. These supported phones can be accessed as an array; a particular phone is chosen by index number. The `GetPhoneInfo()` function will take an index number and return a structure containing the information on the phone from the array of phones supported by the telephony server. This information structure is defined by the `TPhoneInfo` struct given below:

```
struct TPhoneInfo
{
    TNetworkType iNetworkType;
    TName iName;
    TUint iNumberOfLines;
    TUint iExtensions;
};
```

The information given in this structure specifies the type of network of which the phone is a part (an enumerated type, specifying mobile, analog, or digital), the name of the phone, the number of lines supported by the phone, and the number of extensions the phone TSY module supports.

TSY information can be obtained from a telephony server using the functions below:

```
IMPORT_C TInt GetTsyName(const TInt aIndexOfPhone,
   TDes &aTsyName) const;
IMPORT_C TInt IsSupportedByModule(const TDesC &aTsyName,
   const TInt aMixin, TBool &aResult) const;
IMPORT_C TInt GetTsyVersionNumber(const TDesC &aTsyName,
   TVersion &aVersion) const;
```

The name of the TSY that supports the phone at a certain spot in the phone list is retrieved by the first function. The name of the TSY for the phone at the position given in the first argument is retrieved into the second argument. The second function returns a Boolean value that determines if extended functionality is supported by a TSY. All TSY modules are assumed to support a minimal set of telephony functionality. The `IsSupportedByModule()` module will set the Boolean variable in the third argument to reflect if the TSY in the first argument supports the extra functionality specified in the second argument. The second argument is a bitmask of extended functionality constants. Finally, the version number of the TSY module is retrieved by the third function.

Building Your Own TSY

While this book does not cover the construction of custom TSYs, it is worth mentioning that there are plenty of clues as to what it takes in Symbian OS SDK. By browsing files such as `et_tsy.h` and even `et_phone.h` in the `\Symbian\6.1\Quartz\Epoc32\Include` directory, you can gain an appreciation for the kind of implementation that must be done. These files contain the TSY base classes and comprise part of the structure needed for a TSY.

A client of the telephony server can designate itself as a client of highest priority by using the function below:

```
IMPORT_C TInt SetPriorityClient() const;
```

Only one client can be designate itself as a high priority client.

Let's take an example. Let's say we have a `PhoneCall` class that will initialize a phone and make a voice phone call. The definition for this class might look like this:

```
class PhoneCall : public CActive
{
   enum TCallState {EDialing, EDone, EError};
public:
   PhoneCall();
   ~PhoneCall();

public:
   // Static construction
   static PhoneCall* NewLC();
   static PhoneCall* NewL();

public:
   void ConstructL();

   void Init();
   void MakeCall(TDesC &aTelephoneNumber);

   void DoCancel();
   void RunL();

private:
   RTelServer telserver;
   RPhone gsmphone;
   RLine phoneline;
   RCall phonecall;

   TRequestStatus callstatus;
   TCallState iCallState;
};
```

Notice that this class is an active object and uses the `iCallState` variable to track its communication state and the `callstatus` variable to monitor its I/O progress.

Now, consider the definition of the `Init()` function as it applies to the telephony server. Here, we want to deal with a voice call over a GSM phone.

```
void PhoneCall::Init()
{
   RTelServer::TPhoneInfo phoneinfo;
   RPhone::TLineInfo lineinfo;
   RPhone::TCaps capabilities;
   RLine::TCaps lcapabilities;
   TInt result;
   TInt phones,lines,calls;
   TBuf<25> name;

   // Connect to the telephony server
   result = telserver.Connect();
   User::LeaveIfError(result);

   // Load the right TSY
   result = telserver.LoadPhoneModule(_L("gsmbsc.tsy"));
   User::LeaveIfError(result);

   // Get information about phones from the server
   result = telserver.EnumeratePhones(phones);
   User::LeaveIfError(result);
   if (phones == 0) User::LeaveIfError(KErrNotSupported);

   // ... other code to init phones, lines, and calls
}
```

We connected to the telephony server and loaded the GSM TSY module. If no phones were supported (for example, if no GSM phone could be found via the infrared port), this code would abort with an error code.

14.3.2 The Phone Component

When access to the telephony server is granted, a phone supported by the telephony server must be selected. Phones are characterized by the `RPhone` class. This class definition is long; we will take a piecemeal walk through it.

RPhone class functions

Once a session has been established with the telephony server, a subsession with a phone must be established through an `RPhone`

object. As you might expect, we use `Open()` and `Close()` functions for this:

```
IMPORT_C TInt Open(RTelServer &aSession,
   const TDesC &aName);
IMPORT_C void Close();
```

The session variable for an established telephony server session and the name of a phone with which to open a subsession is passed to the `Open()` function. This name is retrieved from `RTelServer` object's `GetPhoneInfo()` function.

When a phone subsession has been successfully opened, the phone must be initialized. This is done through an `Initialise()` function as below:

```
IMPORT_C TInt Initialise();
IMPORT_C void Initialise(TRequestStatus &aStatus);
```

This function is available in synchronous (the first version) and asynchronous (the second version) varieties. The asynchronous version also has a function to cancel the initialization:

```
IMPORT_C void InitialiseCancel();
```

Initializing is allowed to be asynchronous, because it may take some time to set up the telephony device (this is especially true of a modem).

When a phone subsession exists and the device has been initialized, there are several queries that we can make of a telephony device:

- We get the device's capabilities through the following function:

  ```
  IMPORT_C TInt GetCaps(TCaps &aCaps) const;
  ```

  These capabilities are returned in the `TCaps` structure. This structure comprises a bitstring, whose bit positions denote a specific capability. These capabilities include voice transmission, the ability to transmit fax data using various fax standards, and the ability to automatically sense when data is transferred over the modem interface.

- The status of the telephony device can be retrieved using the following function:

  ```
  IMPORT_C TInt GetStatus(TStatus &aStatus) const;
  ```

This information is most useful for a modem. The `TStatus` structure holds information about the status of a connection: if one has been made and how far that connection has progressed.

- Information about the phone lines supported by a particular `RPhone` object can be obtained using the following functions:

```
IMPORT_C TInt EnumerateLines(TInt &aCount) const;
IMPORT_C TInt GetLineInfo(const TInt aIndex,TLineInfo
                          &aLineInfo) const;
```

Following the pattern of the `RTelServer` class, `EnumerateLines()` will return a count of the lines supported on the `RPhone` object queried and `GetLineInfo()` will return a structure with the information about a phone line connected to it. The `TLineInfo` structure is defined as follows:

```
struct TLineInfo
{
  RCall::TStatus iStatus;
  TUint32 iLineCapsFlags;
  TName iName;
};
```

It contains the status of the call going across a line, the capabilities of that line, and the name of the line.

- Finally, information on the phone itself—the same information that was available from the `RTelServer` object—is retrieved using the function below:

```
IMPORT_C TInt GetInfo(TPhoneInfo &aPhoneInfo) const;
```

Look Up These Constants

While I am certainly breezing through all these structures and constants, complete definitions can be found in `\Symbian\6.1\Quartz\Epoc32\Include\etel.h`. You are *urged* to examine this file as the complete reference point for ETel definitions.

When using a telephone device, changes in capabilities or functionality can occur. Symbian OS provides two functions to notify applications about any changes.

```
IMPORT_C void NotifyCapsChange(TRequestStatus &aStatus,
   TCaps &aCaps);
IMPORT_C void NotifyCapsChangeCancel() const;
```

```
IMPORT_C void NotifyModemDetected(TRequestStatus &aStatus,
   TModemDetection &aDetection);
IMPORT_C void NotifyModemDetectedCancel() const;
```

Each function is asynchronous and comes with a corresponding cancellation function. The first notifier signals the caller when the telephony device's capabilities change; the second signals when a modem change has been detected. These functions are best used by callers that are active objects themselves, so that device notification can be combined and watched along with other notification functions (e.g., I/O notification or line changes).

More on the `PhoneCall` example

Let's continue the `PhoneCall` class example. We need to expand the implementation of the `Init()` function to encompass initializing phones. The result is below:

```
void PhoneCall::Init()
{
   RTelServer::TPhoneInfo phoneinfo;
   RPhone::TLineInfo lineinfo;
   RPhone::TCaps capabilities;
   RLine::TCaps lcapabilities;

   TInt result;
   TInt phones,lines,calls;
   TBuf<25> name;

   // ...code to initialize the telephony server connection

   // Get the information on the phone we need
   result = telserver.GetPhoneInfo(0, phoneinfo);
   User::LeaveIfError(result);
   name.Copy(phoneinfo.iName);

   // Open the phone and get its capabilities
   result = gsmphone.Open(telserver, name);
   User::LeaveIfError(result);
   result = gsmphone.GetCaps(capabilities);
   User::LeaveIfError(result);
   if (capabilities.iFlags & RPhone::KCapsVoice == 0)
        User::LeaveIfError(KErrNotSupported);

   // ... other code to init lines and calls
}
```

Note that the phone is automatically initialized when opened. So we do not need to call `Initialise()` from this initialization

code. In the code above, we retrieve the name of the first phone from the telephony server and open it up. On a Nokia 8290 phone (a GSM phone used in the United States), the name of this first phone is `GsmPhone1`. We conclude this code by making sure that the phone we obtained can indeed support voice capability.

14.3.3 The Line Component

Once a subsession with a phone has been established, we need to establish a subsession over a particular line. The line implementation is captured by the `RLine` class. Again, since the definition is large, I will pick it apart and look at it in pieces.

As with `RPhone` objects, `RLine` object subsessions are opened and closed. The functions to do this are below:

```
IMPORT_C TInt Open(RPhone &aPhone,const TDesC &aName);
IMPORT_C TInt Open(RTelServer &aServer,
   const TDesC &aName);
IMPORT_C void Close();
```

Notice that we can open a session using either an `RTelServer` object or an `RPhone` object. The name of the telephony line to use must also be given; this should match a line name retrieved from the `RTelServer` or `RPhone` object being used.

Once a subsession has been established, we can make several queries of an `RLine` object:

- We can get a line's capabilities the same way we did with an `RPhone` object, using the same `TCap` structure:

```
IMPORT_C TInt GetCaps(TCaps &aCaps) const;
```

- We can get the status of a line using two functions:

```
IMPORT_C TInt GetStatus(RCall::TStatus &aStatus) const;
IMPORT_C TInt GetHookStatus(RCall::THookStatus
                            &aHookStatus) const;
```

  The first function returns the status of a call that might occur over the line. This status is an enumeration that depicts a call as idle or in one of a number of states of making a call—from ringing through answering to hanging up. The hook status is a more general view of a call, returning an enumeration that indicates either "on hook" (meaning a disconnected line) or "off hook" (meaning a call in progress).

- Information about calls opened on a line can be retrieved using the functions below:

```
IMPORT_C TInt EnumerateCall(TInt &aCount) const;
IMPORT_C TInt GetCallInfo(TInt aIndex,
   TCallInfo &aCallInfo) const;
```

  As with other telephony objects, the first function gives the number of calls allowed on a line. The second function gets information on a specific call. The `TCallInfo` structure is defined below:

```
struct TCallInfo
{
    TName iCallName;
    RCall::TStatus iStatus;
    TUint32 iCallCapsFlags;
};
```

  This gives the name of the call, its status, and its capabilities.

- Finally, information about a line itself can be obtained using the function below:

```
IMPORT_C TInt GetInfo(TLineInfo &aLineInfo) const;
```

  This information includes the status of the current call (if there is one) and the name of the last call that was made on this line.

As with `RPhone` objects, `RLine` objects can be notified when aspects of a line change. There are many different properties that can change, and this is reflected in the number of notification functions that are defined for the `RLine` class. These are below:

```
IMPORT_C void NotifyIncomingCall(TRequestStatus &aStatus,
   TName &aName);
IMPORT_C void NotifyIncomingCallCancel() const;
IMPORT_C void NotifyHookChange(TRequestStatus &aStatus,
   RCall::THookStatus &aHookStatus);
IMPORT_C void NotifyHookChangeCancel() const;
IMPORT_C void NotifyStatusChange(
   TRequestStatus &aStatus,RCall::TStatus &aLineStatus);
IMPORT_C void NotifyStatusChangeCancel() const;
IMPORT_C void NotifyCallAdded(TRequestStatus &aStatus,
   TName &aName);
IMPORT_C void NotifyCallAddedCancel() const;
```

There are four notification functions, each with its own cancellation function. Each is also asynchronous and requires a status variable for monitoring. In addition, each function has a second parameter that will be filled in by the telephony server when notification is made. The `NotifyIncomingCall()` function will notify when an incoming call is detected and will return the name of the incoming call (calls have names—see the next section). `NotifyHookChange()` will notify when there has been a change in the hook status of a line and the current hook status is returned. The `NotifyStatusChange()` function will notify interested parties about changes in line status, returning a `TStatus` structure from the `RCall` class that reflects the status of a line. Finally, the `NotifyCallAdded()` function will notify when a call is added to the line. This occurs when a new call has been created (see next section) and its properties are added to the line in question. Adding a call does not necessarily mean that a phone call has actually been made, just that the call object is associated with the current line.

Let's continue to flesh out the `PhoneCall` class example. Initializing a line for a phone means getting its name and opening a subsession, as below:

```
void PhoneCall::Init()
{
   RTelServer::TPhoneInfo phoneinfo;
   RPhone::TLineInfo lineinfo;
   RPhone::TCaps capabilities;
   RLine::TCaps lcapabilities;

   TInt result;
   TInt phones,lines,calls;
   TBuf<25> name;

   // ... code to init telephony server and phone

   // Get the info on the line we need
   result = gsmphone.GetLineInfo(2, lineinfo);
   User::LeaveIfError(result);
   name.Copy(lineinfo.iName);

   // Open the line and get its capabilities
   result = phoneline.Open(gsmphone, name);
   User::LeaveIfError(result);
   result = phoneline.GetCaps(lcapabilities);
   User::LeaveIfError(result);
   if (lcapabilities.iFlags & RLine::KCapsVoice == 0)
            User::LeaveIfError(KErrNotSupported);

   // ... code to init call
}
```

This example chooses the third line available on the phone and checks its capabilities. On my Nokia 8290 phone, this third line is the voice line (the first two are fax and data), and the name of this line is `Voice`.

14.3.4 The Call Object

Having established a session with the telephony server and opened subsessions to a phone and a line, we can finally open and manage a call. Calls are described by the `RCall` class. Before we launch into how to use this class, I should point out a few things about calls.

- *Calls have names.* As with other telephony components, a call has a name. The name of a call is typically generated by the OS and returned when a call subsession is created. A "fully qualified" call name is one that includes call, line, and phone information. in the format `PhoneName::LineName::CallName`.
- *Opening a call subsession does not connect a call.* As with phones and lines, a call subsession must be opened before we can use a call. Opening a subsession allows the telephony server to allocate memory and resources for a call but does not manipulate the call in any way.
- *Calls can be incoming as well as outgoing.* In addition to instructing the telephony server to make a call, we can instruct the server to answer a call. This is especially useful when receiving faxes.

Opening a call subsession

A call subsession must be opened before the call can be manipulated. A new subsession is opened with the following functions:

```
IMPORT_C TInt OpenNewCall(RTelServer &aServer,
   const TDesC &aName,TDes &aNewName);
IMPORT_C TInt OpenNewCall(RTelServer &aServer,
   const TDesC &aName);
IMPORT_C TInt OpenNewCall(RPhone &aPhone,
   const TDesC &aName,TDes &aNewName);
IMPORT_C TInt OpenNewCall(RPhone &aPhone,
   const TDesC &aName);
IMPORT_C TInt OpenNewCall(RLine &aLine,TDes &aNewName);
IMPORT_C TInt OpenNewCall(RLine &aLine);
```

A new call can be opened by referencing an open telephony server session, a phone subsession or a line subsession. If a server session

is referenced, the fully qualified name of the call must be given so that the OS can derive the phone and line to use. If a phone subsession is referenced, the line name must be passed. If a line is referenced, there is no need to further specify anything. Notice that some calls come with a second parameter; this parameter is filled in with the name of the call if it is provided.

Subsessions can be opened with existing calls, i.e., calls in progress. This is done by applications that want to work with calls already received or started by other applications. For example, if one application placed a voice call, a second application could implement a call timer. To hang up after a certain time period, the timer application would have to open the existing call. We can do this using the functions below:

```
IMPORT_C TInt OpenExistingCall(RTelServer &aServer,
   const TDesC &aName);
IMPORT_C TInt OpenExistingCall(RPhone &aPhone,
   const TDesC &aName);
IMPORT_C TInt OpenExistingCall(RLine &aLine,
   const TDesC &aName);
```

In all cases, the second parameter must contain the name of the call to open. As with the `OpenNewCall()` function, the format of the name will change depending on the type of reference made. For a telephony server reference, for example, the name must be fully qualified as `PhoneName::LineName::CallName`.

As an example, we can complete the `PhoneCall::Init()` function. Here, we simply open a new call subsession by referencing the line subsession (the simplest way):

```
void PhoneCall::Init()
{
   RTelServer::TPhoneInfo phoneinfo;
   RPhone::TLineInfo lineinfo;
   RPhone::TCaps capabilities;
   RLine::TCaps lcapabilities;
   TInt result;
   TInt phones,lines,calls;
   TBuf<25> name;

   // ... code to init server, phone, and line

   // Open a new call
   result = phonecall.OpenNewCall(phoneline, name);
   User::LeaveIfError(result);
}
```

On my Nokia phone, this call is assigned the name `VoiceCall1`.

Making a call

Although it may seem like a long journey, eventually all sessions and subsessions will be opened and initialized. At this point, we still have not made a call, but the system is ready for this next step.

Calls are made in one of two ways. They can be dialed by the Symbian OS device or they can be connected to an already dialed call. To dial a call, we would use one of the following functions:

```
IMPORT_C TInt Dial(const TTelNumberC &aTelNumber) const;
IMPORT_C TInt Dial(const TDesC8 &aCallParams,
   const TTelNumberC &aTelNumber) const;
IMPORT_C void Dial(TRequestStatus &aStatus,
   const TTelNumberC &aTelNumber);
IMPORT_C void Dial(TRequestStatus &aStatus,
   const TDesC8 &aCallParams,
   const TTelNumberC &aTelNumber);
```

The first two functions are synchronous; the second two are asynchronous. All functions take a parameter of the type `TTelNumberC`—a typedef for a `TDesC` string. There are also two functions—one in each pair—that includes a specification of call parameters. This is a generic byte buffer specific to each TSY module. Note that if the asynchronous dial functions are used, we can use the following function to cancel the dialing operation:

```
IMPORT_C void DialCancel() const;
```

To connect to a previously dialed call, the following functions may be used:

```
IMPORT_C TInt Connect() const;
IMPORT_C TInt Connect(const TDesC8 &aCallParams) const;
IMPORT_C void Connect(TRequestStatus &aStatus);
IMPORT_C void Connect(TRequestStatus &aStatus,
   const TDesC8 &aCallParams);
```

As with the dialing functions, connection functions come in synchronous or asynchronous varieties and can include call parameters. If an asynchronous connection call was used, we can cancel it using the following function:

```
IMPORT_C void ConnectCancel() const;
```

To illustrate this, let's define the `MakeCall()` function from our `PhoneCall` example. For this definition, I have decided to make the

`PhoneCall` class an active object, and I can use an asynchronous version of the `Dial()` function:

```
void PhoneCall::MakeCall(TDesC &aNumber)
{
   phonecall.Dial(callstatus, aNumber);
   iCallState = EDialing;
   SetActive();
}
```

Since we have already set up the telephony system, this is a simple implementation. The number is a string and we return from this function right away while the system dials the call. When the call is dialed, the status variable will change state and the active object's `RunL()` function will be called. We can field this change by including the following code in the `RunL()` function:

```
switch (iCallState) {
   case EDialing:
      if (callstatus == KErrNone) {
         // handle the successful call
      } else {
         // handle the call error
      }
      break;
```

This is just like the active object code we have seen before.

Answering a call

If we have successfully created all the sessions and subsessions we need, answering an incoming call is straightforward. One of the functions below should be used:

```
IMPORT_C TInt AnswerIncomingCall() const;
IMPORT_C TInt AnswerIncomingCall(
   const TDesC8 &aCallParams) const;
IMPORT_C void AnswerIncomingCall(TRequestStatus &aStatus);
IMPORT_C void AnswerIncomingCall(TRequestStatus &aStatus,
   const TDesC8 &aCallParams);
```

The first two versions are synchronous; the second two are asynchronous. Each pair has one version that gives a general buffer variable set by the system when a call comes in. The asynchronous versions of this function can be cancelled with the function below:

```
IMPORT_C void AnswerIncomingCallCancel() const;
```

Once a call has been answered, the application is free to do what it wants with the open phone call. If this is a data call, the application

might want to access the data port directly for a time. For example, if we are making a call with a modem to transfer some data, we might want the telephony server to take care of dialing the phone number and connecting to the opposite side, but we will then want control to pass our data. We do this by "loaning" the data port to the application using the following functions:

```
IMPORT_C TInt LoanDataPort(TCommPort &aDataPort) const;
IMPORT_C void LoanDataPort(TRequestStatus &aStatus,
   TCommPort &aDataPort);
```

This function comes in synchronous and asynchronous versions. Both versions return information on the data port that is involved in the loan. The `TCommPort` structure contains the CSY module's filename and the name of the data port itself. As usual, if the asynchronous version is used, we can cancel the loan by using the function below:

```
IMPORT_C void LoanDataPortCancel() const;
```

Once the data port has been used for the transmission of data (e.g. using sockets), it can be returned to the telephony server using the function below:

```
IMPORT_C TInt RecoverDataPort() const;
```

Notifications and call information

For a call, there are a few notifications that the system can give an application. The intent to receive these notifications can be registered or cancelled with the following functions:

```
IMPORT_C void NotifyHookChange(TRequestStatus &aStatus,
   THookStatus &aHookStatus);
IMPORT_C void NotifyHookChangeCancel() const;
IMPORT_C void NotifyStatusChange(TRequestStatus &aStatus,
   TStatus &aCallStatus);
IMPORT_C void NotifyStatusChangeCancel() const;
IMPORT_C void NotifyCallDurationChange(
   TRequestStatus &aStatus, TTimeIntervalSeconds &aTime);
IMPORT_C void NotifyCallDurationChangeCancel() const;
```

There are three changes in status that can be signaled to an application: the hook status, the call phase status, and the call duration. The hook status is the status of a call's connection (on-hook/off-hook). Remember that the remote device or the switching network can terminate a call at any time without asking for permission or giving

prior warning. The call phase status is a representation of the state of a call as it passes through its "life cycle": idle, dialing, ringing, answering, connecting, connected, or hanging up. The call duration is the time, in seconds, that the call has been active. This is notified to a registrant every second (useful to know as most calls carry a connect charge based on the duration of the call).

Information about the duration of a call—the time between off-hook and on-hook status changes—can be derived from the following call:

```
IMPORT_C TInt GetCallDuration(
   TTimeIntervalSeconds &aTime) const;
```

Complete information about a call in progress can be obtained through the `GetInfo()` function:

```
IMPORT_C TInt GetInfo(TCallInfo &aCallInfo) const;
```

This function returns a structure that depicts information about the call:

```
class TCallInfo
{
  public:
    IMPORT_C TCallInfo();
    TName iCallName;
    TName iLineName;
    THookStatus iHookStatus;
    TStatus iStatus;
    TTimeIntervalSeconds iDuration;
};
```

This can be derived at any time for an active call subsession.

Call ownership

To avoid problems with multiple clients accessing the same call, Symbian OS designates a specific client as the *owner* of a call. This ownership is initially passed to the client that connected to a phone call first, but it can be transferred to another client. For example, one contact manager client may be responsible for setting up a data call while another client is responsible for the actual data transfer.

The transfer of call ownership is managed by the following functions:

```
IMPORT_C TInt TransferOwnership() const;
IMPORT_C void AcquireOwnership(
   TRequestStatus &aStatus) const;
IMPORT_C void AcquireOwnershipCancel() const;
```

The owner of a call signals its desire to transfer the ownership by calling `TransferOwnership()`. A potential receiver of that ownership transfer calls `AcquireOwnership()`; this call is asynchronous and blocks until the `TransferOwnership()` function is called. Cancellation of the `AcquireOwnership()` function is done through `AcquireOwnershipCancel()`.

The ownership status of a call can be retrieved by the following function:

```
IMPORT_C TInt GetOwnershipStatus(
   TOwnershipStatus &aOwnershipStatus) const;
```

The call returns a variable that is an enumeration specifying the ownership.

Terminating a call

Call termination, that is, "hanging up" the phone, is accomplished through the functions below:

```
IMPORT_C TInt HangUp() const;
IMPORT_C void HangUp(TRequestStatus &aStatus) const;
```

These functions will terminate the call immediately. The asynchronous function can be cancelled using the function below:

```
IMPORT_C void HangUpCancel() const;
```

14.4 A Special Case Phone Call: Faxing

A fax call is a phone call with some special properties. It is a data call with some structure (faxing is done in "page" units). Special support for faxing is built into Symbian OS.

Faxing is supported through the `RFax` class. The definition of this class is below:

```
class RFax : public RTelSubSessionBase
{
public:
  IMPORT_C RFax();
  IMPORT_C TInt Open(RCall& aCall);
  IMPORT_C void Close();

  IMPORT_C void Read(TRequestStatus &aStatus,TDes8 &aDes);
  IMPORT_C void Write(TRequestStatus &aStatus,
     const TDesC8 &aDes);
```

```
    IMPORT_C void WaitForEndOfPage(
       TRequestStatus &aStatus) const;
    IMPORT_C TInt TerminateFaxSession() const;

    IMPORT_C TInt GetProgress(TProgress &aProgress);
     ...
};
```

A fax subsession is based on a call subsession. The normal way to handle this is to create a call subsession and to connect using this call (either by initiating or answering a call). We would then perform a fax operation by using the established call object in the `RFax::Open()` function. This establishes the fax data protocol with the remote side.

Once the fax call is established, faxes can be sent or received using the `Read()` and `Write()` functions. These functions take image data and transfer that data using fax protocols. These functions are asynchronous.

The end of a received page can be retrieved through the `WaitForEndOfPage()` function. Because this function is also asynchronous, the `Read()` function can be used in parallel with this function (using different status variables) to read data but terminate images as the ends of pages arrive.

The fax session can be terminated with the `TerminateFaxSession()` function. This function terminates the fax protocol but not the call itself.

Information about faxing can be manipulated in two ways. First, the `RFax` class above has a function to retrieve information about a fax in progress. A call to the `GetProgress()` function will return a `TProgress` structure, defined below:

```
struct TProgress
{
    TTime iLastUpdateTime;
    TBuf<20> iAnswerback;
    TFaxPhase iPhase;
    TFaxResolution iResolution;
    TFaxCompression iCompression;
    TInt iECM;

    TInt iPage;
    TInt iLines;
    TInt iSpeed;
};
```

This structure gives all the information available about a fax-in-progress.

Fax settings are retrieved and set through the `RCall` class. The two functions below work with these:

```
IMPORT_C TInt GetFaxSettings(
   TFaxSessionSettings &aSettings) const;
IMPORT_C TInt SetFaxSettings(
   const TFaxSessionSettings &aSettings) const;
```

These functions get and set the settings that govern fax calls: the fax protocol class, the speed of the transfer, the resolution and compression settings, and so forth.

14.5 Summary

In this chapter, we have taken a look at how Symbian OS supports telephony. Symbian OS views telephony in terms of four components: a telephony server, a phone, phone lines defined for a phone, and finally a call made over a phone line. We reviewed how to set up the structure so that we can use phone calls, and we looked at making and answering calls. We defined how applications can be notified when changes in the phone system and settings are made and we reviewed faxing as a special case of a phone call.

The next chapter will focus on a very important part of Symbian OS: messaging. Messages are at the heart of communication, and Symbian OS has built a large yet flexible infrastructure around them. We will look at this infrastructure next.

15

Sending and Receiving Messages

In everyday life, messages can take many forms. There are verbal and non-verbal messages; messages are written on paper and heard via audio devices; messages can be notes passed in secret or signs on a billboard. Delivery of messages can sometimes be a chancy thing: if you have ever had to rely on a family member to deliver a telephone message, you know what I mean! Even with this wide assortment of message types and delivery functions, humans are able to send and receive these messages fairly easily. We have, in fact, developed an internal system that processes different message types using the same processes, implemented by a tool specialized to each message.

As we have seen, electronic messaging is also a very diverse area. There are many message types and delivery takes many different forms (e.g., delivering email over a PPP network connection is very different than sending an SMS message to a mobile phone). So it should come as no surprise that designing a single framework that will characterize and work with all message forms is quite a challenge. The designers of Symbian OS took on this challenge with EPOC Release 5 and its email system and sharpened their skills through subsequent Symbian OS releases. In the current release of the OS, they have built a messaging system that can not only handle several disparate message types but can accommodate future messaging needs as well.

This chapter overviews the message framework of Symbian OS. We start with an overview—a survey of messaging components and requirements. We then begin looking at the message system API by examining the message server that manages the message store resource. We will then examine the classes that characterize messages themselves. We follow this by looking at "send-as messaging", which forms the heart of the ability to send messages, and "watchers", the functionality for receiving messages. We will wrap

up the chapter by looking at four specific message types and how the messaging system deals with them.

15.1 Overview of the Architecture

Let's start by stepping back for a moment and taking an overview of how Symbian OS views messaging. As they built the messaging system, the designers were guided by the need to put together a generic framework that could handle the components of many different message types. They built a framework that viewed all messages as composed of generic components. Through the use of object orientation, each message type is handled by a separate implementation of those generic components. Symbian OS message architecture brings these implementations together under a common messaging application and a common API.

As we look at messages and their components, it is important to note where these messages come from. All Symbian OS devices have a central message store. This store contains all messages received by and created on the device on which it resides and has a specialized, hierarchical format (as we will explain later in this chapter). It can reside on any storage accessible to a Symbian OS device; message applications can change where this store resides. Originally, it resides on the C: drive of the emulator device.

Now let's look at the messaging framework by pulling apart a message and then building the framework from its component parts.

15.1.1 Dissecting a Message

Back in Chapter 5, we looked at messages as having several common characteristics. Figure 15.1 depicts these characteristics and adds a few more. Messages are composed of *delivery information* and *content* and are generally characterized by *message types*.

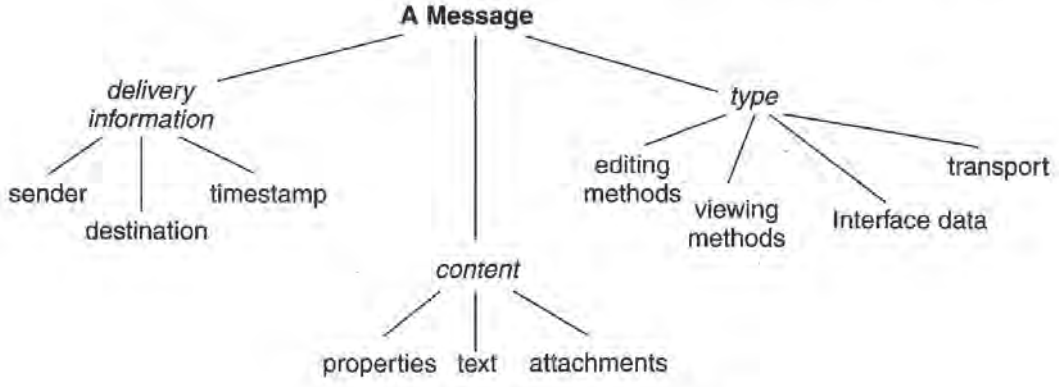

Figure 15.1 Message characteristics.

Delivery information is composed of sender information, destination information, and timestamp information. We reviewed these components back in Chapter 5 and gave several examples. Email messages, for example, contain all this information textually in the header of a message. Sender information is contained in the `Received:` and `From:` fields; destination information is kept in the `To:` field; timestamp information is found in the `Date:` field.

Content has three parts: properties, message content and possibly a set of data objects or attachments. An email message can again serve as a straightforward example. There are pieces of message header that specify certain properties of the message; the `Content-Type:` and `Content-Length:` fields, for example, indicate the MIME properties and the length of the body. The message itself is contained in the body of an email message as text. In email, data objects can also appear in the message body as MIME attachments.

Each message also has a *message type*. The type of a message is the definition of a larger class of messages that describes general characteristics of that class. A message type cannot describe the specific contents of a message, but it can describe properties of the message class. These properties include:

- *The editing function:* This property is a description of how to edit a message of a particular message type. On a Symbian OS device, this "description" takes the form of an implementation of a message editor application.
- *The viewing function:* This property describes how to view a message of a particular message type. Again, in Symbian OS, this "description" is actually an implementation of a message viewer application.
- *User interface data:* There can be certain data associated with user interfaces that deal with a certain message type. These data include items like icons and progress dialog interfaces to display for the message type.
- *The transport function:* Messages of a certain type are transported in the same way to and from their destinations or repositories. The implementation of this transport is associated with the message type and not each individual message.

Let's look at email messages again. All messages that are of the "email" class of messages might be viewed the same way in Symbian OS—through an email viewer that can present the textual message with tools to view header information as well as any attachments to the email. There is a common composition (editing) interface that you can use to compose new messages or edit draft

messages. There is a set of user interface definitions—some icons for email applications to display, for example—that can be accessed. Finally, email is transported to a server via SMTP and from a server via POP3 or IMAP4. (Since there are three transport functions, one could argue that there are actually three message types for email. Indeed, Symbian OS views it this way as well.)

Now if we expand our view to other types of messages, we can see that the framework established for Symbian OS will work for any type of message we have encountered. Fax, SMS, and BIO messages all have this format, albeit with different implementations for each component. For that matter, BTIM messages from Chapter 13 could fit right into this framework but would require new implementations for API components.

BTIM Messages in Symbian OS Message Structure

Consider BTIM messages from Chapter 13. These are simple text messages that shuttle back and forth between devices over Bluetooth. This type of messaging is a prime target for incorporation into the Symbian OS messaging framework.

Consider how this might work. Instead of a separate application, we would integrate BTIM messaging into the existing framework and work with BTIM messages using the messaging application on a Symbian OS device.

Consider what this would look like. How would this look in the application? What viewer or editor might be necessary? What interface information would you need? What delivery information might be required? As we move toward designing our own BTIM components, we need to answer these basic questions.

15.1.2 Putting the Pieces Together: MTMs

Symbian OS uses *message type modules* (MTMs) to define message types. An MTM is composed of four classes that are used as base classes for specific message handling implementations. These four classes are as follows:

- The *user interface MTM* is defined as the `CBaseMtmUi` class and defines user interface capabilities, such as viewing and editing messages.
- The *client-side MTM* is defined as the `CBaseMtm` class and handles the interface between the internal representation of a message's data and the user interface MTM.

- The *user interface data MTM* is represented by the `CBaseMtmUi-Data` class and provides access to user interface data properties (e.g., icons).
- The *server-side MTM* is defined in the `CBaseServerMtm` class and provides message transport capability.

Each message type, then, has an implementation of each of these MTMs, subclassing their definitions from the classes above. Some message types can share MTMs; IMAP email and POP email, for example, can share the same user interface MTM, while having different server-side and client-side MTMs.

15.1.3 Some Perspective on this Architecture

This way of modeling messages is powerful and very effective. Through various MTM implementations, Symbian OS supports its four main message formats (email, SMS, fax, and BIO formats) as well as several formats that work behind the scenes to implement other protocols.

It is interesting to examine the disparity between the various forms of messaging under the MTM framework. Fax messaging, for instance, is vastly different from SMS messaging, yet both fit comfortably in this model. Although faxing has different delivery requirements than SMS and must be viewed as an image, where SMS can be viewed as text, both can still be integrated with the same messaging application on a Symbian OS device.

The large number of different forms of messaging results in a large number of MTM implementations. Any messaging application must be able to sift through these implementations to find the necessary MTMs as quickly as possible. Symbian OS helps in this endeavour by providing a registry of installed MTMs that are accessible to an application. This registry allows MTM components to be identified and instantiated quickly and easily.

Now consider this structure from a programmer's perspective. Imagine you are a programmer who wants to implement the sending of an SMS message as a small part of a larger application. It would be quite daunting for you to use the MTM framework in all its glory to send a "simple" SMS message. In fact, you might just reject the idea as not worth the time and effort. The designers of the MTM model understood that a complicated structure might frustrate programmers who want to do simpler tasks, so they added something to the architecture called *send-as messaging*. Send-as messaging provides a simple interface that allows applications to create outgoing messages by using a single API. By using send-as

messaging, a programmer uses one interface for any message and simply informs the OS what type of message to send; the OS implementation will figure out which MTMs to use and how to send the message. This is a powerful idea and it reinforces the modularity of the MTM structure. It has proven to be an easy and effective interface. It is so effective, in fact, that many applications use it to transport data other than traditional messages. The Agenda engine, for example, uses send-as messaging to send vCalendar objects over the IR interface.

Interesting MTM Implementations

Using the Find function in your Windows Start button, look for file names that include "MTM" in your installation of Symbian OS SDK. You will find several that might look familiar, but you will also find some that do not. For example, there is an MTM module implemented as a DLL called `WAPPUSHMTMCLIENTSERVER.DLL`. This is an implementation for messages that would show up in the messaging applications that include WAP formatted content.

Look for other messaging types that do not look familiar, and consider what they might be used for.

15.2 The Message Server

15.2.1 Message Server Functionality

Symbian OS considers the data repository for messages as a resource that can be shared between processes. Therefore, following Symbian OS standards, there is a server that protects this repository and manages access to it. The message server implements two valuable functions:

- *Access to the message data repository:* As clients request message access, the message server delegates temporary, exclusive access to message data. The message server must keep the message data repository correctly ordered and must cope with anomalous message events correctly. For example, access failures or incomplete sessions must not corrupt the message data.
- *Access to MTMs:* The message server must enable applications to identify requests, such as the sending of a message, that require protocol-specific functionality and load the appropriate MTM.

The message server accepts requests from client applications that require access to messages and MTMs. These requests are handled asynchronously, as they might require a large amount of time to perform (fax messages, for example, typically take much more time to process than email messages). Requests can include changing the structure or contents of the local folders, sending and receiving messages through different services, changing the structure or contents of remote mailboxes, and MTM-specific requests. Note that actions such as delivery and storage concepts such as local folders and remote mailboxes are all managed by the message server.

15.2.2 Storage and Message Structure

The Symbian OS message store has a specific structure shown in Table 15.1. The diagram in Figure 15.2 shows an overview of this structure. Messages are stored in folders, which are accessed as children of services. A service can be viewed as a message source; an ISP is a service, as is local, on-device storage. Folders can be user-defined or system-defined; for example, in and out boxes are defined by the system and are applicable under the local storage service. Folders can have subfolders, which can have subfolders, etc. This hierarchical organization of the storage structure means that access to messages requires a walk through the storage tree before access is granted to a message.

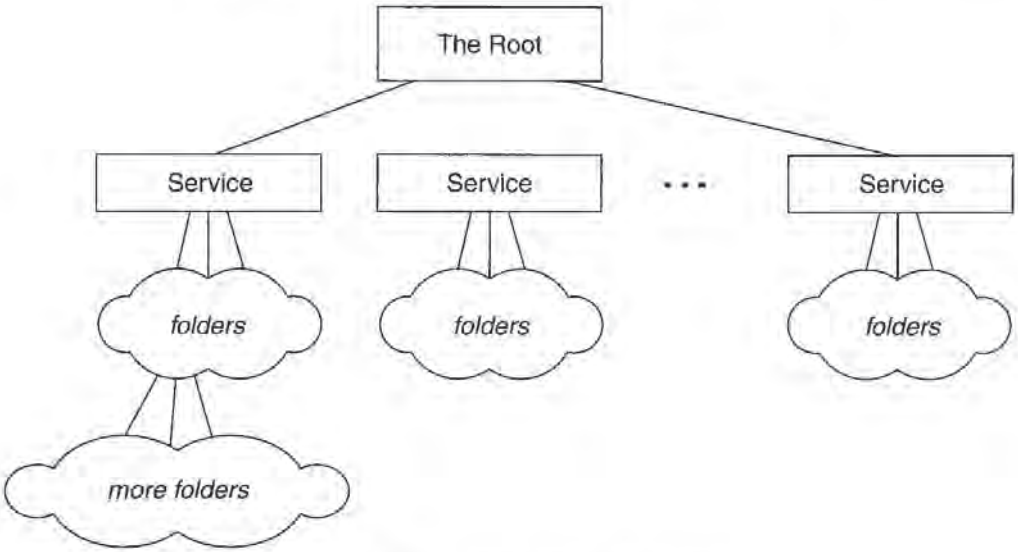

Figure 15.2 Message storage structure

Messages themselves also have a structure. This structure is shown in the table below.

Table 15.1 Message structure

Message component	Type of data
Generic message header	Message index entry data
MTM-specific information	MTM-stream data
Message body	Rich text data in message store.
Attachments	Attachment files stored in the message's store

Message index entries have a specific format. That format is quite large, and I will not reproduce it here. Suffice it to say that both delivery and content header information is stored here and that this information varies for different MTMs. The MTM-stream date is specifically formatted for the type of MTM that is used for the message. Obviously, this information also varies for the MTMs. The message body is a rich text object (from the `CRichText` class) that contains text as well as formatting information. Details about attachments are found in the message header—the index portion—but the data files are stored in their own format in the message store after the message body.

15.3 Accessing Messages

To access messages, we must create and manipulate several objects. First, we must create a session with the message server. Next, we must initialize contact with the message system registry in such a way that we can create MTMs later to handle messages. Finally, we access the message system by walking through the message structure tree: we first access the system root, then find the message's service, then the message's folder (and possibly subfolders), and then we finally access a message.

15.3.1 Creating a Session with the Message Server

A session with the message server is encapsulated in an object from the `CMsvSession` class. The class is large, so we will dissect slowly. As a first step, let's examine the functions that open a session with the server:

```
static CMsvSession* OpenSyncL(
   MMsvSessionObserver &aObserver);
static CMsvSession* OpenAsyncL(
   MMsvSessionObserver &aObserver);
```

The first function opens a session synchronously, i.e., when the function call returns, the session has been started and is valid. The second

function opens a session asynchronously. When the session is valid, the caller is informed through the `MMsvSessionObserver` object. Notice that both functions require an observer object derived from the `MMsvSessionObserver` class. This class is a base class and using it mandates that you create a subclass that uses this one as its foundation. Specifically, the subclass must implement the single function in this class:

```
void HandleSessionEventL(TMsvSessionEvent aEvent,
   TAny* aArg1, TAny* aArg2, TAny* aArg3);
```

This function will be called as events happen in the session that was created. The `TMsvSessionEvent` argument contains the event that caused the function call, and the `TAny` arguments contain any data that will be used with the event to provide more information. We will deal with this observer functionality later in this chapter.

15.3.2 Opening the Registry

The next step is to open the registry so that we can obtain MTM objects when we need them. There are three kinds of access into the registry: one to access client-side MTMs, one to access user interface MTMs, and one to access user interface data MTMs. Each access is implemented by its own class: `CClientMtmRegistry`, `CMtmUiRegistry`, and `CMtmUiDataRegistry` classes, respectively. They are derived from the same base classes and have similar definitions.

No Server-side MTM Access

Accessing the registry does not give us access to server-side MTMs. Because all outgoing functionality is encapsulated by send-as messaging, access to server-side MTMs is funneled through this system. We will see this later.

Let's look at the `CClientMtmRegistry` class as a representative. It has the following definition:

```
class CClientMtmRegistry : public CObserverRegistry
{
public:
  IMPORT_C static CClientMtmRegistry* NewL(
     CMsvSession &aMsvSession, TTimeIntervalMicroSeconds32
     aTimeoutMicroSeconds32=TTimeIntervalMicroSeconds32(30000000));
  IMPORT_C ~CClientMtmRegistry();
  IMPORT_C CBaseMtm* NewMtmL(TUid aMtmTypeUid);
  //
   ...
};
```

There is a function to create a new instance, a function to destroy an instance, and a function to create a new MTM handle. The `NewL()` function takes an active message server session object and an optional timeout parameter (without this parameter, the call times out in 30 seconds). The `NewMtmL()` function takes the UID of the type of MTM we will need. Each MTM has a UID, as shown in Table 15.2:

Table 15.2 MTM UIDs for Symbian OS version 6.1

MTM Type	UID Value	Constant Name
Text	0x10005247	`KUidMsgTypeText`
SMTP	0x10001028	`KUidMsgTypeSMTP`
POP3	0x10001029	`KUidMsgTypePOP3`
IMAP4	0x1000102A	`KUidMsgTypeIMAP4`
Fax	0x1000102B	`KUidMsgTypeFax`
SMS	0x1000102C	`KUidMsgTypeSMS`

Other MTM Registries

You can find out about the other MTM registry interfaces by browsing their definitions. The `CClientMtmRegistry` class is defined in the `mtclreg.h` file; the `CMtmUiRegistry` class is found in the `mtuireg.h` file; and the `CMtmUiDataRegistry` class is found in the `mtudreg.h` file. These definition files are found in the `\Symbian\6.1\Quartz\Epoc32\include` directory.

To make an instance of a registry class and create an MTM handle for later use, we could use the code below:

```
CClientMtmRegistry ClientMTMRegistry;
CBaseMtm* ClientMtm;

ClientMTMRegistry = CClientMtmRegistry::NewL(
   MsgSvrSession);
ClientMtm = ClientMTMRegistry ->NewMtmL(KUidMsgTypeText);
```

Notice that we had to give a message server session object (already established in the `MsgSvrSession` variable) and the type of message for which we will need the MTM. In the example above, we are intending to use client-side MTMs for text messages.

15.3.3 Accessing the Message Tree

Once we have established a valid message server session and we have access to the registry, we can access the message tree structure.

Entries in this tree structure are represented by the `CMsvEntry` class. All types of entries, from root entries through to messages themselves, are represented by this class. Entries are retrieved from the message tree by using functions from the `CMsvSession` class. Two functions from that class are relevant here:

```
inline CMsvEntry* GetEntryL(TMsvId aEntId);
inline TInt GetEntry(TMsvId aId, TMsvId &aService,
   TMsvEntry &aEntry);
```

The `GetEntryL()` function looks up a message tree entry by the entry's ID. In the first version of the function, the call returns a pointer to a `CMsvEntry` class object that represents the message being retrieved. In the second version of the function, other information is returned: the service of which the entry is a child and the header information (the index entry, a `TMsvEntry` object) of the entry we are retrieving. The entry ID of the root of the message tree is given by the constant `KMsvRootIndexEntryId`.

Let's stop for a moment and look at the characteristics of an entry in the message tree as represented by the `CMsvEntry` class. An entry includes the following properties:

- *an entry ID*, an object of the `TMsvId` class, retrieved by the `CMsvEntry::EntryId()` function
- *the header information*, what we earlier referred to as entry index information, obtained via the `CMsvEntry::Entry()` function
- *an "owning service"*, i.e., the parent of an entry, retrieved by the `CMsvEntry::OwningService()` function, which returns an entry ID
- *a certain number of children* (for the root and inner tree nodes), accessed by a number of functions, e.g., `CMsvEntry::Children()` and `CMsvEntry::Count()` functions
- *file storage*, determined by the Boolean `CMsvEntry::HasDirectoryL()` and `CMsvEntry::HasStoreL()` functions and retrieved by the `CMsvEntry::GetFilePath()` function
- *an MTM list*, set mostly by the system as entries are created

Consider an example. Let's say we need a function that would look up a node and walk through each of its children. Consider the following code:

```
void MessageExample::EntryWalk(TInt aEntryId)
{
   TInt i;
   TInt iCount;
   CMsvSession *iMessageSession;
```

```
    CMsvEntrySelection* iChildren;
    CMsvEntry* iRootEntry;
    CMsvEntry *child;

    // Open a session with the message server
    iMessageSession = CMsvSession::OpenSyncL(*this);
    iRootEntry = iMessageSession->GetEntryL(aEntryId);
    iCount = iRootEntry->Count();

    // Get the root of the hierarchy
    iChildren = iRootEntry->ChildrenL();
    for (i=0; i<iCount; i++) {
       child = iMessageSession->GetEntryL((*iChildren)[i]);

       // do something with the child

    }
}
```

In this code, we open a session to the message server, get the node entry referenced by the `aEntryId` parameter, find its children, and then loop through fetches of each child. If we wanted to look at each service that the message server manages, we could call this function with the entry ID of the root of the message tree (`KmsvRootIndexEntryId`). Since services are direct children of the message tree root, this would access each service. We could repeat the process (perhaps recursively) to eventually reach each message. As another example, if we wanted to look at each message in the message tree inbox for the local message store, we would call this function with the entry ID of `KMsvGlobalInBoxIndexEntryId`, which is defined as the inbox ID constant.

Notice in the above code that the children of an entry are retrieved into a `CMsvEntrySelection` class. This class is defined as an array of entry IDs.

15.3.4 Viewing and Editing Messages

Once we have a message entry as a `CMsvEntry` object that we want to manipulate, we need to get its interface MTM to view it or edit it. We can follow this sequence to obtain and use this MTM:

- Obtain the header information from the entry. This will be in the form of a `TMsvEntry` object.
- Use the data from the header information to discover the MTM type that handles the entry. For example, the UID of the MTM can be found as the `TMsvEntry::iMtm` variable of this object.
- Request instances of the client-side MTM and user interface MTM objects from the registry.

- Set the context of the client-side MTM. This means that we "focus" the client-side MTM on a specific message, so that it can configure itself to obtain the right types of data (e.g., MTMs).
- Call the operation function (i.e., either viewing or editing) provided by the user interface MTM. This returns an active object to manage the asynchronous operation.
- When that operation object completes, it will signal the active object implemented by the client program.
- That active object then handles any action, such as cleanup, that is required.

From this sequence, notice that the actual operation of viewing or editing is an asynchronous operation, governed by an active object. This object is an instance of the `CMsvOperation` class, which is derived from the `CActive` class (among others). This class is used extensively by `CMsvEntry` objects and user interface MTMs to implement functions that require a parallel active object thread and have the possibility of cancellation. In addition to creation and cancellation of the `CMsvOperation` object, several other operations are possible. The definition below is taken from the definition of the `CMsvOperation` class:

```
virtual const TDesC8& ProgressL()=0;
IMPORT_C virtual const TDesC8& FinalProgress();
IMPORT_C virtual TUid Mtm() const;
inline TMsvOp Id() const;
inline TMsvId Service() const;
```

The progress and final disposition of the operation performed (when it completes) are available in human-readable strings from the first two functions. The `Mtm()` function will return the UID of the MTM that is being used for the operation. The `Id()` function returns the ID of the operation internal to the message server. The ID allows a client to keep track of different operations. Finally, the `Service()` function returns the entry ID of the service that is involved in the operation.

Now, let's follow an example that implements the above sequence. Assume we have a `CMsvEntry` message entry object as the variable `child`, from the example above. We would view this message as follows:

```
TRequestStatus status;
TMsvEntry *msginfo;
CClientMtmRegistry iMTMClientRegistry;
CMtmUiRegistry iMTMUiRegistry;
CBaseMtm *iClientMtm;
CBaseMtmUi *iMtmUi
```

```
msginfo = child->Entry();

// Create a client-side MTM
iMtmClientRegistry = CClientMtmRegistry::NewL(
   iMessageSession);
iClientMtm = iMtmClientRegistry->NewMtmL(msginfo->iType);

// Set the client MTM's context
iClientMtm->SwitchCurrentEntryL(child->EntryId());

// Extract the text message MTM UI
iMTMUiRegistry = CMtmUiRegistry::NewL(iMessageSession);
iMtmUi = iMTMUiRegistry ->NewMtmUiL(iClientMtm);

// View the message
iMtmUi->ViewL(status);
User::WaitForRequest(status);
```

In this code, we create a new client-side MTM by using the registry to obtain one for the type of message we are looking at, as derived from the child message. We then set the *context* of the client-side MTM—i.e., customize it—to the current message. At this point, the MTM interface object is focused on the right message and the right context/message type, and we can view the message through that interface object.

15.3.5 Other Operations on Messages

The definition of the `CBaseMtmUi` class is lengthy. It includes functions for viewing and editing messages (as we have just seen), copying and moving messages between folders, replying to and forwarding messages, deleting messages, and monitoring the progress of operations. In all cases, since these operations may take a while, the functions work like `EditL()` and `ViewL()` above, i.e., they invoke a `CMsvOperation` active object and return a handle to it.

Just as operations for viewing and editing messages are tailored for a specific message type, replying and forwarding messages are also implemented differently for each message type. You would reply to a fax differently than you would to an SMS message. These are handled by asynchronous functions, as in the definitions below:

```
virtual CMsvOperation* ReplyL(TMsvId aDestination,
   TMsvPartList aPartlist,
   TRequestStatus &aCompletionStatus) = 0;
virtual CMsvOperation* ForwardL(TMsvId aDestination,
   TMsvPartList aPartList,
   TRequestStatus &aCompletionStatus) = 0;
```

Both functions take an entry ID of the service or folder where the newly-created message from a successful completion should

be placed. For example, using the supplied constant, `KMsvGlobalOutBoxIndexEntryId`, would place the resulting new message in the outbox to await delivery. The second parameter to these functions is a "parts list"—an integer whose bit positions determine which parts of the original message will be included in the new message. Message parts are defined by the constants that are OR'd together. The final parameter is a status variable used to monitor the progress of the active object.

Messages and services can be deleted through the functions below:

```
IMPORT_C virtual CMsvOperation* DeleteFromL(
   const CMsvEntrySelection &aSelection,
   TRequestStatus &aStatus);
IMPORT_C virtual CMsvOperation* UnDeleteFromL(
   const CMsvEntrySelection &aSelection,
   TRequestStatus &aStatus);
IMPORT_C virtual CMsvOperation* DeleteServiceL(
   const TMsvEntry &aService, TRequestStatus &aStatus);
```

The first two functions take a selection array and a status indicator. They allow messages to be deleted or undeleted from folders. The third function will delete a service, including the folders and messages that are dependent on that service.

There are four functions that implement the transfer of messages to and from remote servers. They are defined below:

```
IMPORT_C virtual CMsvOperation* CopyToL(
   const CMsvEntrySelection &aSelection,
   TRequestStatus &aStatus);
IMPORT_C virtual CMsvOperation* MoveToL(
   const CMsvEntrySelection &aSelection,
   TRequestStatus &aStatus);
IMPORT_C virtual CMsvOperation* CopyFromL(
   const CMsvEntrySelection &aSelection,
   TMsvId aTargetId, TRequestStatus &aStatus);
IMPORT_C virtual CMsvOperation* MoveFromL(
   const CMsvEntrySelection &aSelection,
   TMsvId aTargetId, TRequestStatus &aStatus);
```

The first two—`CopyToL()` and `MoveToL()`—will copy or move messages indicated by the selection array to the current context. The context is the current folder to which you have navigated in your application prior to executing one of the above functions. Since the messages will have their own context information encoded in each reference in the selection array, the system will know the context from which to transfer the message. The second two functions—`CopyFromL()` and `MoveFromL()`—do not have this

information, because they transfer messages from the current context to some other context. You must provide the entry ID of that destination context in the `aTargetId` argument.

Behind the Scenes

Stop for a moment and consider what is going on behind the scenes of remote message transfer. The implementation of this will be different for different message types.

Consider a session with a remote email server. The local device's copy or move operations will access the server through the existing socket and will pass commands to the server through this socket. These operations do not need additional connections as one already exists.

Two functions allow information to be obtained about on-going operations. Consider the definitions below:

```
IMPORT_C virtual TInt DisplayProgressSummary(
   const TDesC8 &aProgress) const;
IMPORT_C virtual TInt GetProgress(
   const TDesC8 &aProgress,
   TBuf<EProgressStringMaxLen> &aReturnString,
   TInt &aTotalEntryCount, TInt &aEntriesDone,
   TInt &aCurrentEntrySize,
   TInt &aCurrentBytesTrans) const;
```

These functions convert progress information about some asynchronous operation, information that is specific to an MTM such as message sending, to a human-readable form. The progress information that you must pass into these functions is a string obtained from an on-going `CMsvOperation` by calling its `ProgressL()` function. The nature of any information that is displayed is MTM-specific. The first function simply displays a progress message. The second function actually gets more information so you can display it in your application.

Read Up on the Definitions

Because of the size of these class definitions, I have brushed over several other definitions. There are other class definitions and constant specifications that you should review. Specifically, you should look at the definition for the `CMsvOperation` and `TMsvPartsList` classes, among others.

15.3.6 MTM-Specific Functions

In the implementations of different MTMs, there are bound to be functions that are specific to each MTM that do not generalize to all MTMs. For example, an email message MTM that uses POP3 would need functions to connect to and disconnect from the POP3 server. A BIO message MTM would not need a function to connect to a server because there is no server for BIO messages (they are pushed to a device). On the other hand, a BIO message MTM needs a function to process the BIO message content. This applies to SMS and fax messages as well. Each MTM implementation will require its own specialized functions.

The messaging architecture provides a way for MTM components to offer protocol-specific functionality not provided by base class interface functions. These MTM-specific functions are implemented in the MTM and assigned IDs that correspond to each protocol-specific operation offered. To call these specific functions, each MTM component implements two functions, called `InvokeSyncFunctionL()` and `InvokeAsyncFunctionL()`, that allow clients to access these operations by passing in the appropriate ID. As you can guess from their names, the first function allows an operation to be called synchronously and the second allows asynchronous access to an operation. The actual calling sequence is defined below from the `CBaseMtm` class:

```
virtual CMsvOperation* InvokeAsyncFunctionL(
    TInt aFunctionId, const CMsvEntrySelection &aSelection,
    TDes8 &aParameter,
    TRequestStatus &aCompletionStatus) = 0;

virtual void InvokeSyncFunctionL(TInt aFunctionId,
    const CMsvEntrySelection &aSelection,
    TDes8 &aParameter) = 0;
```

Each call requires the function ID as the first parameter. This is different for each MTM used. The second parameter defines the messages that are affected as a message selection array. The third parameter is a generic buffer of byte data that provides a parameter list to the function being invoked. Finally, the asynchronous call requires a status variable for monitoring.

Look Up MTM–Specific Function IDs

You can find the MTM-specific function IDs in certain header files. Under the `\Symbian\6.1\Quartz\Epoc32\include`

directory, look for files whose names end with `cmds.h` (e.g., `imapcmds.h` for IMAP functions). These contain a list of functions with function IDs defined by a C++ enumeration.

15.4 Creating Basic Messages: The `BaseMtm` Class

As we have discussed, Symbian OS supports several kinds of message types and allows new types to extend the architecture. While each message type has its own MTM and its own set of progress monitors, there is a core structure that should look familiar. All MTMs are derived from a single base MTM class: the `CBaseMtm` class. Each MTM subclass will inherit functions from the base class and will customize them for its own purposes. We will go over the core set before elaborating on the derived classes.

The first functions we should review are functions that determine the context of an MTM: `SetCurrentEntryL()` and `SwitchCurrentEntryL()`:

```
IMPORT_C void SetCurrentEntryL(CMsvEntry* aEntry);
IMPORT_C void SwitchCurrentEntryL(TMsvId aId);
inline CMsvEntry& Entry() const;
inline TBool HasContext() const;
```

These functions change and test the entry on which later actions are performed through the MTM. The first two functions are distinguished by their parameter types. The first takes a pointer to a service, folder, or message; the second takes an entry ID number. A message client application must use one of these to set a context before using other MTM functions. The third function retrieves the context of the MTM, and the last function tests if one has been set.

When the context is set, no data is manipulated. The context is the *environment* of the message—its representative properties—but not its *contents*. Message data must be explicitly loaded and saved through the functions defined below:

```
virtual void SaveMessageL()=0;
virtual void LoadMessageL()=0;
```

To retrieve entry data, invoke the `LoadMessageL()` function after setting the context; any changes made to a message should be saved by calling `SaveMessageL()`. There is a cache that is set up for the message, which is emptied upon switching contexts; calling these functions will fill and save this cache.

A message can be created—i.e., its entry is created and the message framework established—by the `CreateMessageL()` function. This is defined below:

```
IMPORT_C virtual void CreateMessageL(TMsvId aServiceId);
```

The parameter is an entry ID for the service to which this message will belong.

Specific parts of a message can be manipulated on their own using the `CBaseMtm` class. The functions include the ones below:

- Addresses can be added to and deleted from the recipient lists of messages through the functions below:

  ```
  inline const CDesCArray &AddresseeList() const;
  virtual void AddAddresseeL(
     const TDesC &aRealAddress) = 0;
  virtual void AddAddresseeL(
     const TDesC &aRealAddress, const TDesC &aAlias) = 0;
  virtual void RemoveAddressee(TInt aIndex) = 0;
  ```

  Addresses are simply strings for these function calls, and the address list is an array of strings. Addresses are removed by using indexes from this address list array. These functions are designed to allow address list manipulation with no MTM-specific knowledge.

- The subject of a message is controlled through the functions defined below:

  ```
  IMPORT_C virtual void SetSubjectL(
     const TDesC &aSubject);
  IMPORT_C virtual const TPtrC SubjectL() const;
  ```

  These functions will set the subject and retrieve the subject. The subject itself is simply a string.

- The body of a message can only be retrieved in the `CBaseMtm` base class. Subclasses will define ways to set the body on their own. Retrieving the body of a message is accomplished by functions defined below:

  ```
  inline CRichText& Body();
  inline const CRichText& Body() const;
  ```

  Note that these functions return a `CRichText` object. A `CRichText` class object is generic enough to handle all the different formats that a body could be in for the various types of messages we will use.

The body of a message is cached by the operating system—even after manipulation—so that access to it can be fast. To move the cache to and from file storage, we need to use one of two functions, defined below:

```
IMPORT_C void StoreBodyL(CMsvStore &aStore);
IMPORT_C void RestoreBodyL(CMsvStore &aStore);
```

`StoreBodyL()` and `RestoreBodyL()` encapsulate the retrieval and storage of this `CRichText` object to a `CMsvStore` for implementers of derived classes. If you want to read or write the body without reading or writing the entire message, this is how to do it.

- Attachments are important for several types of messages. We manipulate attachments by the following functions:

```
IMPORT_C virtual void CreateAttachmentL(
   TMsvId &aAttachmentId, TFileName &aDirectory);

IMPORT_C virtual void DeleteAttachmentL(
   TMsvId aMessageId, TMsvId aAttachmentId);
```

  These functions do not actually create or delete anything more than an index entry to be filled in by the application. When `CreateAttachment()` is called, an index entry is created for the attachment, and the entry ID for it is returned along with the directory in which to create the attachment. Deleting an attachment with `DeleteAttachmentL()` must use the entry ID for a message and for the attachment ID on that message.

Without a specific type of a message to use, it is hard to give a concrete example of the above functions. But we can review the sequence of events that must occur to create a message.

- First, we must create the MTM for a message type. We would do this as we did before using the registry. Let's assume we have an MTM called `theMTM`, created with a `KUidMsgTypeSMTP` message type (email using SMTP).
- Next, we would have to create a message. Let's say we want to create a message in the local message service's outbox. We would use code like this:

```
theMTM->CreateMessage(KmsvGlobalOutBoxIndexEntryId);
```

- Next, we would want to set up the destination of the message. For example, we might use the code below:

```
theMTM->AddAddresseeL(_L("jipping@cs.hope.edu"));
```

  This sets the message to go out to my email address.
- We could set up the subject as follows:

```
theMTM->SetSubjectL(_L("Meeting at 3:00 pm"));
```

- We would then manipulate the message body as the specific type of message MTM stipulates.
- If we want to manipulate attachments, we would do so with the functions from the specific MTM.
- We can now save the message. We could use the code below:

```
theMTM->SaveMessageL();
```

  This saves the message to the outbox.

Notice how once the message has been created—and the context set by that creation—all future function calls work on that message context. There is no need in the calls to specify which message to manipulate.

15.5 Easy Sending of Messages: Send-As Messaging

By now you might be a little taken aback by the complexity of message handling and the message architecture of Symbian OS. While it is true that the architecture is big and complex—after all, it must handle messages as different as faxes and SMS messages–the designers of Symbian OS have streamlined the process we use to send messages. The sending process is relatively short and very straightforward. Let's review that process in this section.

The sending procedure is called *send-as messaging*. It is a generic process—i.e., one process for all message types—that uses a message type's MTM to guide the sending process. It is a powerful model that can be used to send messages of all types and even to transfer data in unique ways. The procedure follows the sequence we outlined in the last section.

Send-as messaging centers on the `CSendAs` class. The definition of this class is rather large, so we will not go through it in its entirety. Rather, we will go through the process necessary to send a message and overview the API along the way.

15.5.1 Step 1: Choose the MTM

The first step is to choose the MTM that the `CSendAs` object will use to send the message. This can be done in two ways: we can set the MTM directly or we can use a `CSendAs` object to search for the MTM we need.

The first function is the easiest, but we must know (a) which MTM we want and (b) that the MTM is available on the device we are using. To set the MTM, we need the `CSendAs` function defined below:

```
IMPORT_C void SetMtmL(TUid aMtmUid);
```

For this function, we need to know the UID of the MTM we intend to use. There are constants defined for this purpose for the installed MTMs; Table 15.2 has some of the predefined standard UIDs.

The second way to set the MTM is to search for it. To use this function, we need to give the `CSendAs` object some search criteria to help it prune the list of available MTMs. To do this, we would use following function:

```
IMPORT_C void AddMtmCapabilityL(TUid aCapabilityId,
   TBool aResponseExpected=EFalse);
```

We would add each of the necessary capabilities, which would filter the MTM list for those that have the capabilities we need. In version 6.1 of Symbian OS, there are 15 predefined UIDs for MTM capabilities. For example, if we wanted an MTM that supports subjects and attachments, we might use the code below:

```
iSendAs->AddMtmCapabilityL(KUidMtmQuerySupportSubjects);
iSendAs->AddMtmCapabilityL(
   KUidMtmQuerySupportAttachments);
```

Once the capability list has been set, the list of available MTMs can be accessed through the function defined below:

```
inline const CDesCArray& AvailableMtms() const;
```

This function returns an array of strings that indicate the MTMs that possess the capabilities stipulated.

This list of capabilities can be reset by a call to the following function:

```
IMPORT_C void ResetMtmCapabilitiesL();
```

Once an MTM has been chosen from the list of available ones, the MTM can be set for a `CSendAs` object by using an index into the array of available MTMs, using the function below:

```
IMPORT_C void SetMtmL(TInt aMtmIndex);
```

This function is an overloaded alternative to the `SetMtmL()` function we discussed above.

15.5.2 Step 2: Choose the Service to Use

Once the MTM has been chosen, we need to set the service to use for the outgoing message. To do this we use the two functions below:

```
inline const CDesCArray& AvailableServices() const;
IMPORT_C void SetService(TInt aServiceIndex);
```

The first function will return an array of strings that describe the services that are available. The second function chooses a service by selecting an index from this array of available services.

If no service is explicitly selected, a default service will be used for sending the message.

15.5.3 Step 3: Create the Message (and All Its Parts)

We begin a message by creating it through the `CSendAs` object. We would use one of the functions described below:

```
IMPORT_C void CreateMessageL();
IMPORT_C void CreateMessageL(TMsvId aId);
```

In both cases, a message for the current `CSendAs` object is created. In the first version, a blank message is created; in the second version, a message is created whose context and contents are copied from the message whose ID is given as a parameter. To create a message, the MTM for the `CSendAs` object must already have been set.

Once the message for the `CSendAs` object has been created, we can create and modify its components. The components are the same ones that we addressed with the `CBaseMtm` class, with similar functions.

- Message recipients are modified with one the following functions:

```
IMPORT_C const CDesCArray& RecipientList() const;
IMPORT_C void AddRecipientL(
   const TDesC &aRealAddress);
```

```
IMPORT_C void AddRecipientL(
   const TDesC &aRealAddress, const TDesC &aAlias);
IMPORT_C void RemoveRecipient(TInt aIndex);
```

The two `AddRecipientL()` functions add the string given to the list of recipients for the message. The list can be retrieved with `RecipientList()` as an array of strings. The `RemoveRecipient()` function removes the recipient at the position in the list indicated by its parameter.

- The subject of the message—as a string—is set with the following function:

```
IMPORT_C void SetSubjectL(const TDesC &aSubject);
```

Obviously, SMS and BIO messages do not have a subject.

- The body of a message is set with a rich text object using this function:

```
IMPORT_C void SetBodyL(const CRichText &aMessageBody);
```

- Any attachments can be modified with the following functions:

```
IMPORT_C void CreateAttachmentL(TMsvId &aAttachmentId,
   TFileName &aDirectory);
IMPORT_C void DeleteAttachmentL(TMsvId aAttachmentId);
IMPORT_C void SetAttachmentL(TMsvId aAttachmentId,
   const TDesC &aAttachmentName);
```

When an attachment is created, its entry ID is returned along with a directory in which that the attachment must be created. Once the file that contains the attachment has been created, that file can be attached by the `SetAttachmentL()` function. We must provide the attachment ID and filename to attach to it as parameters. Finally, attachments can be deleted by using `DeleteAttachmentL()` with the appropriate attachment entry ID.

15.5.4 Step 4: Save and Send the Message

When the message has been prepared, sending the message means saving it into the appropriate service's message store. This means a call to one of the functions below:

```
IMPORT_C void SaveMessageL(TBool aMakeVisible=ETrue);
IMPORT_C void SaveMessageL(TRequestStatus &aStatus,
   TBool aMakeVisible=ETrue);
```

Both versions take a Boolean value, a flag that dictates to the message server whether it is to make the message visible in the message's context. The second version is asynchronous and also takes a status variable through which an application can monitor the progress of the `CSendAs` object. The second version is used for messages that may take more time than an application is willing to wait, e.g., a fax page that must be rendered before being saved.

15.5.5 An Example

Let's consider a `CSendAs` example. Suppose we want to extend the `TodoX` example to send the todo-list item as an attachment to an email message. To make this extension work, it must be a subclass of the `TodoXferBase` class and must implement some meaningless—and therefore empty—functions. But the new class will fit into the transfer scheme nicely. We can also finally transfer a vCalendar object.

In this example, the calling sequence for this new "port" looks like this:

```
iXferPort->Init(_L("jipping@cs.hope.edu"));
iXferPort->OpenL();
iXferPort->Configure();
iXferPort->SendItem(iReader->iEntry);
iXferPort->Close();
```

Here, `iXferPort` is an object of the `TodoXferEmail` class. This looks like all the other calling sequences we have seen for this example.

In order to fit email transfer into the `TodoX` class hierarchy, I had to split up message sending just a bit. If you think of the email message as a "port", this will make more sense. The `Init()` function is defined as follows:

```
TInt TodoXferEmail::Init(TDesC &aRecipient)
{
   TInt result;

   iSendAs = CSendAs::NewL(*this);
   iRecipient.Copy(aRecipient);

   return 0;
}
```

The recipient of the email is sent as the parameter. The `CSendAs` object is created and the recipient stored for future use.

Opening the email message does not make sense here, so the `OpenL()` function has an empty definition.

Configuring the email message means setting the MTM properties of the `CSendAs` object. The definition of `Configure()` is below:

```
void TodoXferEmail::Configure()
{
   if (iSendAs) {
      iSendAs->AddMtmCapabilityL(
         KUidMtmQuerySupportAttachments);
      iSendAs->SetMtmL(KUidMsgTypeSMTP);
   }
}
```

In this code, we make sure that we can support attachments, and we set the message type to be defined as SMTP messages by using the SMTP MTM.

The `SendItem()` function contains the real meat of the sending process. Its definition is below:

```
void TodoXferEmail::SendItem(CAgnEntry *aEntry)
{
   CAgnTodo* iTodoItem;
   TBuf<100> dbuffer;
   TBuf8<100> buffer;
   TUint priority;
   TTime duedate;

   // Generate a file with the vCal object in it
   RFs fs;
   fs.Connect();
   RFile outputFile;

User::LeaveIfError(outputFile.Replace(fs,KOutputFile,
                    EFileWrite));
   RFileWriteStream writeStream(outputFile);
   iAppUi->iReader->WriteVCalObject(aEntry, writeStream);
   writeStream.CommitL();
   writeStream.Close();
   outputFile.Close();

  // Create message
  iSendAs->CreateMessageL(KMsvGlobalOutBoxIndexEntryId);

  if(iSendAs->QueryMessageCapability(
     KUidMtmQuerySupportAttachments) == KErrNone) {
    TMsvId messageId;
    TFileName attachment;
    TFileName srcFile;

    messageId = iSendAs->MessageId()
    iSendAs->CreateAttachmentL(messageId, attachment);
    attachment += KFileName;
```

```
    CFileMan* file = CFileMan::NewL(fs);
    TInt fileCopy = file->Copy(KOutputFile, attachment);
    delete file;

    if (fileCopy == KErrNone) {
      // Save the message, creating an email with the
      // current recording as an attachment.
      iSendAs->SetAttachmentL(messageId, KFileName);
         if    (iSendAs->AvailableServices().Count() < 1)
                User::Leave(KErrNotFound);
      iSendAs->SetService(0);
      iSendAs->AddRecipientL(iRecipient);
      iSendAs->SetSubjectL(_L("A New Todo-List Item"));
      iSendAs->SetBodyL(FormMessageLC(_L
           ("Here is the todo-list item")));
    iSendAs->SaveMessageL(ETrue);
       User::LeaveIfError(iSendAs->ValidateMessage());
   }
   CleanupStack::PopAndDestroy(3); // globalCharLayer,

globalParaLayer, messageBody
   } else {
     //handle the error
   }
}
```

There are several things to point out about this code:

- Notice that we are not actually including the vCalendar object as an attachment, but are including the *text representation* of the vCalendar object as the attachment. The first part of the code creates a file and dumps the vCalendar text from the todo-list item selected into that file.
- After we create an empty message in the message server outbox, we stop to check if the MTM in use can support attachments. We use the `QueryMessageCapability()` of the `CSendAs` object to test this.
- The attachment itself is created by (1) creating an empty attachment with the `CreateAttachmentL()` function, which returns a directory used to store the attachment, (2) appending the filename to this directory name, (3) copying the vCalendar text file to the file indicated by the newly created path name, and (4) telling the message server that the attachment is in the file just copied by using the `SetAttachmentL()` function.
- The rest of the message is created as we discussed earlier: we set the recipient to the previously copied string, we set the subject appropriately, and we set the body of the message to some rich

text object (`FormMessageLC()` returns a rich text object that contains the text specified).

- Finally, saving the message puts it in the outbox, ready for sending.

When the `SendItem()` function is complete, an email message will exist in the outbox folder of the default email service. We can send this message by invoking the message application and using it to send the messages from the outbox, or do this programmatically by using the client-side MTM. The `SendL()` function below defines how to do this:

```
CMsvOperation* TodoXferEmail::SendL(
   TRequestStatus &aStatus)
{
  CMtmUiRegistry* iMtmUiRegistry;
  CBaseMtmUi* iBaseMtmUi;
  CBaseMtm* iClientMtm;
  CMsvSession* iSession;

  // extract MTM information from the SendAs object
  iClientMtm=&(iSendAs->ClientMtm());
  iSession=&(iClientMtm->Session());

  // Establish a connection with the registry
  iMtmUiRegistry=CMtmUiRegistry::NewL(*iSession,10000000);
  iBaseMtmUi=iMtmUiRegistry->NewMtmUiL(*iClientMtm);

  // Create CMsvEntrySelection with a single entry of the
  // message created ClientMtm has context set to last created
  // message, so we use that
  TMsvId entryID=iClientMtm->Entry().EntryId();
  CMsvEntrySelection* selection=new (ELeave)
     CMsvEntrySelection();
  selection->AppendL(entryID);

  // Set context to target service
  // The following reads "the id of the target service in the
  // index entry of the context of the client-side MTM"
  TMsvId serviceID=iClientMtm->Entry().Entry().iServiceId;
  iClientMtm->SwitchCurrentEntryL(serviceID);

  // And send the message by copying it to the service
  CMsvOperation* msvOperation=iBaseMtmUi->CopyToL(*selection,
     aStatus);

  return msvOperation;
}
```

The result of this function is a `CMsvOperation` object, which is an active object that governs the sending process. This active object uses the status variable sent as a parameter to the function for monitoring the progress. This function works by (1) using the

client-side MTM to create a message selection, (2) changing the context to that of the client-side MTM, and (3) copying the message from where it is kept (in our case, the outbox) to the client-side MTM context. This causes the sending operation to initiate and attempt to send out the message selection.

Finally, "closing" the message means deleting the `CSendAs` object. The definition of `Close()` is below:

```
void TodoXferEmail::Close()
{
    delete iSendAs;
    iSendAs = NULL;
}
```

15.6 Receiving Messages

Applications can work with the messaging system and message arrival. While the message server itself handles the actual listening and transport functions, applications can register themselves to be notified upon message arrival.

Registration occurs when a connection with the message server is established. Recall from Section 15.3 that connecting with the message server requires a parameter that is an object of the `MMsvSessionObserver` class. As we said before, this is an abstract class that forces classes derived from it to implement a single function, shown below:

```
void HandleSessionEventL(TMsvSessionEvent aEvent,
   TAny* aArg1, TAny* aArg2, TAny* aArg3);
```

The message server will call this function from the class object passed during the server connection whenever a "message event" occurs. A code that signifies which event actually occurred is passed in the first parameter and up to three pointers to data areas that apply to the message event make up the rest of the function call's parameter list.

Session Event Types

The session events that are possible are defined by the TMsvSessionEvent enumeration. There are 21 different event types defined by this type, and they can be found in `\Symbian\6.1\Quartz\Epoc32\Include\msvapi.h`.

For example, suppose that an application wants to be notified when it receives SMS messages. We might create a message server session as follows:

```
iSession = CMsvSession::OpenSyncL(*this);
```

This means that the object that made this call must be subclassed from the `MMsvSessionObserver` class. Further, we must implement the `HandleSessionEventL()` function. An example of an implementation is below:

```
void TodoXferSMS::HandleSessionEventL(
   TMsvSessionEvent aEvent,
   TAny* aArg1, TAny* aArg2, TAny* aArg3)
{
    TInt i;

    switch(aEvent) {
       case EMsvEntriesCreated:
            // We are only interested in inbox messages.
            // For this event, aArg2 is the parent of
            // the created entries.
            if (*(static_Cast<TMsvId*>(aArg2)) !=
               KMsvGlobalInBoxIndexEntryId) return;

            // aArg1 will be the message selection set
            CMsvEntrySelection* entries =
               static_cast<CMsvEntrySelection*>(aArg1);
            for(i = 0; i < entries->Count(); i++) {
                DoSomething(entries->At(i));
            }
            break;

       default:
         //Ignore all other events.
         break;
    }
}
```

As we are only interested in new messages, we only pay attention to the `EMsvEntriesCreated` event. In addition, we are only interested in new inbox entries, so we check to make sure that the parent folder is the system inbox. When we are sure we have a new inbox entry, we get a list of entry IDs and call the `DoSomething()` function for each ID in the list.

For our example, this `DoSomething()` function must (1) validate that we have an SMS message, and then (2) load and process that message. We can define `DoSomething()` as follows:

```
void TodoXferSMS::DoSomething(const TMsvId &aEntryId)
{
    TInt length;
    HBufC* iMessageData;
    TPtr messageDataPtr;

    iSmsMtm->SwitchCurrentEntryL(aEntryId);

    // Validate the message as an SMS message
    TMsvPartList validationFlags(
                    KMsvMessagePartBody
                  | KMsvMessagePartDescription
                  | KMsvMessagePartOriginator );

   if (iSmsMtm->ValidateMessage(validationFlags) != 0)
      return;
   iSmsMtm->LoadMessageL();
   //Extract the message and notify the observer.
   length = iSmsMtm->Body().Read(0).Length() - 1;
   iMessageData = HBufC::NewL(length);
   messageDataPtr = iMessageData->Des();
   messageDataPtr.Copy(iSmsMtm->Body().Read(0));

   // ... now do something with the message data
}
```

After switching to the context of the message, we validate the message through the SMS MTM by checking the message body, the message's date, and the message's originator/sender. Then we load the message and extract the message body. Finally, we do whatever the application was supposed to do with the message text through the string pointer `messageDataPtr`.

15.7 More Details on Specific Message Types

15.7.1 Email Messages

Email messages are actually handled by three MTMs in Symbian OS. Incoming messages are retrieved by either the POP3 or the IMAP4 MTM; outgoing messages are handled via the SMTP MTM.

The POP3 MTM set is made up of a POP3 client-side MTM, a class that implements email retrieval, a class that holds POP3 settings, and a class that manipulates email entries in the message structure. The client-side MTM, `CPop3ClientMtm`, implements the functionality necessary to treat messages as if they were received using POP3 protocols. It is subclassed from both the `CBaseMtm` and `MEntryObserver` classes. Because of this, all message functions handle messages as if they are POP3 email messages. In addition,

the `MEntryObserver` functionality allows this class to respond to operations on messages from the message server.

The POP3 MTM uses the `CImPop3GetMail` class to retrieve email from a remote host using POP3. It handles connection and disconnection and treats the remote inbox as if it was just another folder under the email service. The POP3 MTM also uses the `CImPop3Settings` class to hold the settings for the POP3 protocol exchange. Properties such as the IP address of the remote inbox server and login name/password for the POP3 protocol are specified with this class. Finally, the `TMsvEmailEntry` class is used by the POP3 MTM to characterize email messages. It is subclassed from the `TMsvEntry` class and tailored to specifically address email messages.

The IMAP4 MTM also consists of four components. Its client-side MTM is implemented by the `CImap4ClientMtm` class and has the same properties as the POP3 client-side MTM. It tailors the `CBaseMtm` functions to specifically address messages retrieved by IMAP4. It also inherits functions from the `MEntryObserver` class, so it can handle events that occur to IMAP4 messages. The MTM uses the functionality built into the `CImImap4GetMail` class to retrieve email using the IMAP4 protocol. The MTM also uses the `TMsvEmailEntry` class to manipulate specific email messages.

The SMTP MTM uses the `CSmtpClientMtm` class as its client-side MTM. This class implements the functionality to send email via SMTP; it handles connection scenarios, delivers messages, and manages the protocol. There is no implementation for the receipt of messages via SMTP. The SMTP MTM uses the `TSmtpProgress` class as a callback class to relay progress information. Progress information for SMTP operations includes the current connection state and the number of messages sent. Settings for SMTP messages and sessions are stored using the `CImSmtpSettings` class. Manipulation of SMTP messages—that is, email messages—is handled through the `TMsvEmailEntry` class as with the other protocols.

15.7.2 SMS Messages

SMS messages are handled by the SMS MTM, which is implemented by a client-side MTM, a settings class, and a progress call-back class.

The client-side MTM is implemented by the `CSMSMtmClient` class. This class provides specific support for SMS messages, including operations to access a device's service information and the scheduled sending of SMS messages. Progress information can be obtained via the `TSmsProgress` class for messaging operations. Progress information includes such things as the type of operation

and the number of messages processed. Settings for SMS messages and delivery are kept and accessed via the `CSmsSettings` class.

SMS messages are typically handled when they arrive but are not explicitly retrieved as email messages are. (Recall the model for message delivery from Chapter 4: SMS messages are delivered via the "push" model.) SMS message receipt is typically done by system telephony and message servers, with received SMS messages placed in the inbox and any session observers notified as a result.

15.7.3 Fax Messages

Fax messages are handled by the fax MTM. This MTM is made up of a client-side MTM, which handles sending and receipt scheduling, along with a progress monitoring class and a settings class.

The fax client-side MTM is implemented by the `CFaxMtmClient` class. This class implements specific operations for sending and receiving faxes and works with new or existing connections. Sending faxes means that outgoing messages are rendered into TIFF images and scheduled for delivery with the telephony server. Received faxes are kept in image format and placed in the message server inbox. While the `CFaxMtmClient` class is subclassed from `CBaseMtm` and `MEntryObserver` classes, some operations are not appropriate for fax message and are not implemented.

Fax message delivery progress is stored and accessed by the `TFaxSessionProgress` class. Information such as the current page being transmitted and the remote device's address are accessible. The MTM settings are accessed via the `TMTMFaxSettings` class.

15.7.4 BIO Messages

BIO messages are handled by Symbian OS using the familiar MTM structure. The BIO MTM is made up of a client-side MTM, a properties set class, and a progress class. In addition, a server-side MTM is an important component.

The client-side MTM is implemented by the `CBIOClientMtm` class. This class has an interface similar to what we have seen before with other client-side MTMs and provides specific support for BIO messages. Progress information can be obtained via the `TBioProgress` class for messaging operations.

The settings for BIO messages must be implemented in a special way. BIO messages are specifically *bearer independent*, that is, they assume no transport function. While BIO can travel over SMS, an infrared transport could just as easily be used. This means that some

kind of structure for settings, transport information, and format must be built. In addition, the use of a BIO message is very open-ended. For example, some messages carry ringtones for phone handsets and some messages carry WAP files for a WAP browser. This structure must be accommodated by the BIO message system. This is complicated by the fact that some BIO settings for specific devices are typically proprietary.

Support for BIO messages and their properties is implemented through BIO Information Files (BIFs) and a BIO message database. BIFs hold BIO message properties and settings; they can be read or written using the `CBioInfoFileReader` and `CBioInfoFileWriter` classes. These files are constructed by the manufacturer of each Symbian device. Manipulation of the BIO database is done through the `CBIODatabase` class. Further analysis of these classes and BIFs are left to the reader.

15.8 Summary

In this chapter, we have discussed the Symbian OS messaging framework. Since messages vary widely in content but can be roughly categorized by form, this framework has been designed to be generic enough to handle all forms of messages but to allow specialization of message handling.

We reviewed the components of a message and outlined the structure of message type modules that implement each message type. Message type modules contain information about how to perform functions, such as editing, viewing and delivering a message for a specific message type. We then discussed the message server, the process that manages messaging stores and facilities, and detailed the ways that we interact with the message server to read and send email. We also discussed "send-as messaging", which is a convenient way to send messages through a simple interface, using functions common to all message types. We reviewed what it takes to communicate with the message server and receive a message. Finally, we reviewed some of the supplied definitions for preinstalled message types in Symbian OS version 6.1.

The next chapter details the HTML and WML content languages and their related delivery protocols.

16

Browsable Content Technologies

The World Wide Web had an inauspicious beginning. In the summer of 1991, Tim Berners-Lee of CERN released his "WWW Program" to the high-energy physics community, where it essentially flopped. He wrote up his work and submitted a paper describing the Web to a conference in 1991. The paper was promptly rejected. Initially written as a way to help his memory, the world of linked hypertext documents took off when Berners-Lee posted an article to the "alt.hypertext" newsgroups and some NeXT machine users got wind of what his idea could really do. Now, more than ten years later, the Web has seen some wild success and has spawned other "browsable" content technologies.

In this chapter, we will take a look at the support that Symbian OS gives to browsable content technologies, specifically HTTP and its small-display descendant WAP. Symbian OS treats HTTP as a member of the TCP/IP protocol family while building new infrastructure to handle elements of the WAP stack. Reviewing HTTP support reinforces our discussion of the TCP/IP stack; examining WAP support provides a good look at how a new technology fits into the Symbian communication architecture.

We will begin this chapter by looking at the support for HTTP in Symbian OS. We will then focus the rest of the chapter on WAP and the support for its suite of protocols in Symbian OS communication architecture. Support for WAP is implemented layer-by-layer, and we will examine each layer.

16.1 HTTP and HTML

To Symbian OS, HTTP is an application protocol supported by the TCP/IP transport infrastructure. There is no special handling of the protocol; there is no special class that encapsulates its functionality.

As a TCP-level protocol, HTTP requires connected sessions between the client and the server, and an implementation of an HTTP client assumes error-free, reliable transmission of data. Recall from Chapter 5 that HTTP is a text-based, stateless protocol that ties a simple client request format with a response from a Web page server in the form of a document typically written in HTML and packaged as a set of MIME objects.

The HTTP stack is shown in Figure 16.1. Symbian OS supports the layers of this stack, although not specifically for HTTP. The layers of the HTTP stack have support in various protocol modules; the `tcpip.prt` module supports the implementation of TCP/IP. There are no specific classes for TCP or IP data packets, because these are handled in the protocol module.

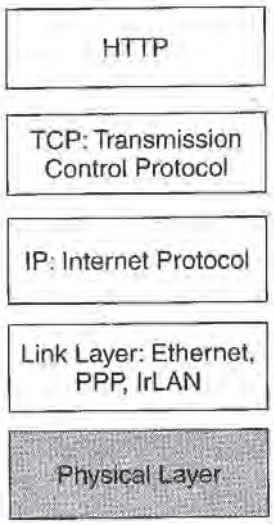

Figure 16.1 Protocol layers that support HTTP.

Transport of HTTP is done via sockets and is handled by the applications that implement the protocol. We demonstrated this type of transport in Chapter 12 with the Web page example.

Web pages are documents which are specified ("marked up") using HTML, a markup language based on SGML. It consists of a generic description of formatting by surrounding text with "tags" that describe the formatting that is needed. The display of HTML documents is done by a browser application that parses the HTML from a file and renders a set of formatted text in a document viewer. Since viewing HTML is an application-based activity, Symbian OS assumes applications will do this and has no built-in support for HTML viewing.

In summary, the transport of Web-based content is an application-based task that is supported by Symbian OS via existing TCP/IP

implementations. The viewing of content is the responsibility of an application, and there is no supporting structure for HTML rendering.

The Web Engine

If you do a bit of poking around in the `\Symbian\6.1\Quartz\Epoc32\Include` files, you will notice references to a Web engine. There are classes devoted to such an engine's implementation and a plug-in infrastructure that is built and supported by header files.

These classes are supplied to support the Web browser that is supplied with Symbian OS version 6.1. It has a plug-in design that takes DLL modules to view documents of special content.

Unfortunately, Symbian OS does not currently provide support for HTML viewers that can be embedded in applications and GUI objects

16.2 WAP

While HTTP is supported as a TCP/IP application protocol, the WAP suite of protocols poses a different problem for Symbian OS. As we discussed in Chapters 4 and 5 (see Figure 4.5 for the WAP stack details), WAP content is supported by a completely different protocol stack. Therefore, an entirely new protocol stack implementation had to be constructed from the bottom up.

16.2.1 Bearers

For WAP suite protocols, bearers constitute the very bottom of the stack, transporting data for the upper layers to use. The bearer layer in the WAP stack is analogous to a combination of the data link layer and the physical communication layer in a generic ISO stack model (see Figure 2.1). Bearers make up the transport infrastructure that gets WAP data packets to the WDP layer for processing. Examples of a bearer is a TCP/IP network; others are SMS messages, and CDMA/CDPD cellular transport.

Symbian OS supports two bearers: TCP/IP and SMS. Symbian OS supports both 7-bit and 8-bit SMS and, within these forms, the OS supports browsing messages and non-browsing messages. Figure 16.2 shows the relationships between the supported data forms. Browsing WAP content is an exercise that typically requires

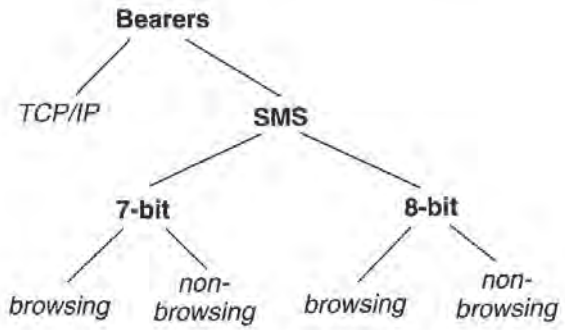

Figure 16.2 WAP bearer types.

quick access to links and rapid-fire downloads. Browsing WAP/SMS types of delivery messages have fewer requirements so as not to slow down the Symbian device with logging and other reporting activities.

It is important to realize here that browsing WAP content is different than browsing HTTP content. HTTP uses a "pull" model, i.e., the client opens a socket to the server, requests information, and the data flows back through the same socket. In a sense, the client reaches through a socket and pulls the information from the server. WAP content is handled differently; browsing WAP content over SMS uses both a "pull" and a "push" model. Although an initial request is sent from the browsing client, it is typical that WAP data will arrive via messages sent from the server to the client. The bearer will determine which type of content delivery is possible. Using a TCP/IP network, for instance, an application could enable a connected "pull" of WAP data; an SMS delivery system would dictate that a connectionless "push" should be used.

16.2.2 WAP Stack Support

Based on experience with transport protocols in Symbian OS, one would guess that support for the WAP stack would take the form of a new socket implementation. All of the other transport functions have been via sockets. However, support for the WAP stack is *modeled after* the socket implementation but is implemented by its own series of classes (analogous to socket types), because it needs to employ both pulling and pushing of data.

Figure 16.3 depicts the WAP stack with the classes that implement various portions of it.

The WAP session server

A WAP session is considered a resource that needs protection and management. So, in standard fashion, Symbian OS uses a server to

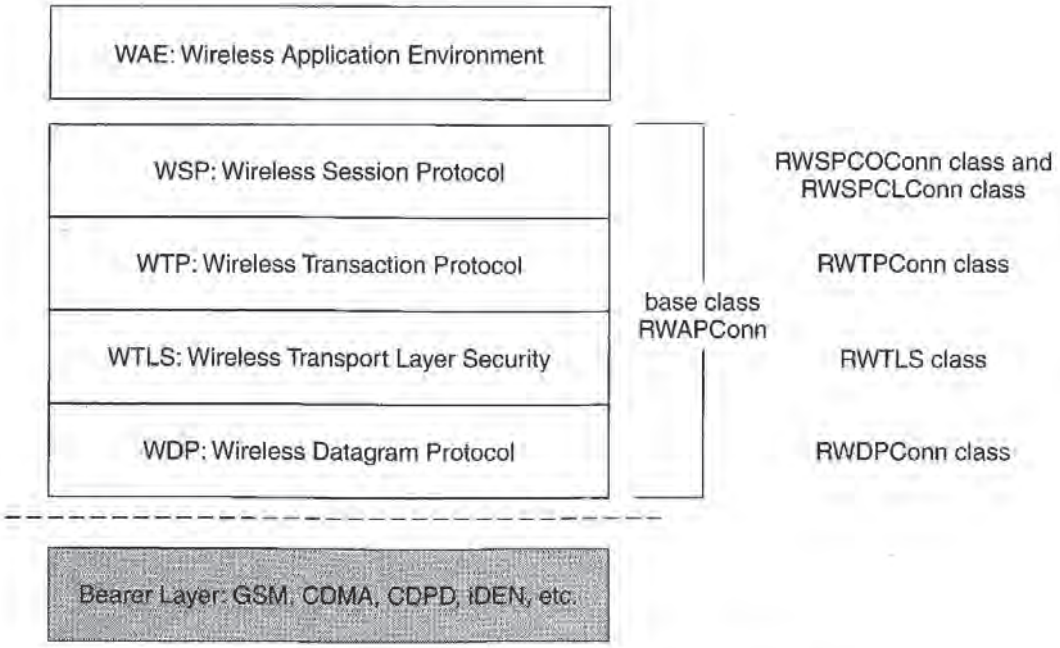

Figure 16.3 Classes implementing the WAP stack.

manage the WAP resource. This server is analogous to the socket server; an application must first connect to the WAP session server before creating and opening a session across another protocol layer. To avoid confusion with the remote WAP server from which an application might retrieve WAP content, I will refer to this server as the *WAP session server.*

The `RWAPServ` class encapsulates the implementation of the WAP session server. The class definition is below:

```
class RWAPServ : public RSessionBase
{
  public:
    IMPORT_C RWAPServ();
    IMPORT_C TInt Connect(TInt aDefaultMessageSlots =
       KDefaultMessageSlots);
    IMPORT_C void ResourceCountMarkStart();
    IMPORT_C void ResourceCountMarkEnd();
    IMPORT_C TInt ResourceCount();
    IMPORT_C TVersion Version() const;
    IMPORT_C void Close();
};
```

The main functions here are `Connect()` and `Close()`; these open and terminate a WAP server session. The `Version()` function will return the current version of the WAP stack implementation. The remaining functions allow us to derive a count of connections to the WAP server; recall that we saw something similar in the SDP

server implementation for Bluetooth. We start counting sessions by calling `ResourceCountMarkStart()`, and we stop counting by calling `ResourceCountMarkEnd()`. Between the two calls, we get the number of sessions using the `ResourceCount()` function.

Once a session with the WAP session server has been established, we can go on to set up sessions through the other protocol layers.

The subsession base class: `RWAPConn`

Other class implementations in the WAP support infrastructure derive definitions from a common base class: `RWPConn`. Through this class, some common concepts are implemented. Its definition is below:

```
class RWAPConn : public RSubSessionBase
{
  public:
    typedef TUint16 TPort;

    IMPORT_C virtual TInt Close();
    IMPORT_C class RWTLS Wtls();

    IMPORT_C void GetRemoteAddress(TDes8 &aAddress, TPort
 &aPort) const;
    IMPORT_C void GetBearer(TBearer &aBearer) const;
    IMPORT_C TInt GetLocalPort(TPort &aPort) const;
    IMPORT_C void CancelAll();

  protected:
    TInt Open(RWAPServ &aServer,
              const TDesC8 &aRemoteHost,
              TPort aRemotePort,
              TPort aLocalPort,
              TBearer aBearer,
              TBool aSecureConn,
              TInt aFunction);
};
```

This class defines the basics for manipulating protocol sessions. We have `Open()` and `Close()` functions to start and terminate protocol sessions. We use `GetRemoteAddress()` and `GetLocalPort()` functions to derive the address and port of the remote partner and local port where the connection is made. There is a `CancelAll()` function that terminates and deletes any pending operations across a protocol session.

To open a WAP protocol session at any level, Symbian OS requires several pieces of information. These are all included in the `Open()` function definition. Note that subclasses of `RWAPConn`

will overload this class definition for their own use. However, this definition contains all the information an `Open()` call needs:

- the WAP session server to use to open the port
- the address and port number of the remote service to which the protocol session is connecting
- the bearer the system is to use to make the connection
- whether or not this is to be a secure connection

If the connection is to be secure, the WTLS layer is created; a handle to this layer can be retrieved by the `Wtls()` function call.

From this base class definition, one can appreciate how closely the socket model has been followed. Protocol sessions are opened and closed like sockets, there are concepts of remote addresses and ports to connect to, we have the notion of canceling (draining) I/O operations before we close up. There are some differences here, namely, the idea of bearers and the optional use of a security layer in the protocol stack.

Implementing the WDP layer

The WDP layer lies just above the bearer layer and provides a general datagram transport service needed by upper layers in the stack. Its implementation is via the `RWDPConn` class. The definition is below:

```
class RWDPConn : public RWAPConn
  {
  public:
    IMPORT_C TInt Open(RWAPServ &aServer,
      const TDesC8 &aRemoteHost,
      TPort aRemotePort,
      TPort aLocalPort,
      TBearer aBearer, // an enum
      TBool aSecureConn);
    IMPORT_C TInt Open(RWAPServ &aServer,
      TPort aLocalPort);

    IMPORT_C void Recv(TDes8 &aRemoteAddress,
      TPckg<TPort> &aRemotePort,
      TDes8 &aBuffer,
      TRequestStatus &aStatus);
    IMPORT_C void CancelRecv();
    IMPORT_C TInt Send(const TDesC8 &aBuffer);
    IMPORT_C TInt SendTo(const TDesC8 &aRemoteAddress,
      TPort aRemotePort,
      const TDesC8 &aBuffer,
      TBearer aBearer);
```

```
    IMPORT_C void RecvSize(
      TPckg<TUint16> &aDataSize,
      TRequestStatus &aStatus);
};
```

First notice that there are two versions of the `Open()` function: fully and not fully specified. The fully specified `Open()` can be used with the `Send()` function; because the information was supplied with the `Open()` function, the `Send()` function does not need destination information. The minimal `Open()` function is paired with the `SendTo()` function, which needs more information about the datagram destination. Note that the `SendTo()` function can be used for multiple destinations, whereas the `Send()` function sends data to only one destination.

To receive a datagram, we need to use the second version of the `Open()` function. The `Recv()` function will accept data but must specify from where the data is coming. The `RecvSize()` function waits for data of a specific length and immediately returns without performing I/O, allowing that data to be picked up by the `Recv()` function.

Finally, note that the sending functions are synchronous, and the receiving functions are asynchronous. The `Recv()` function is paired with a `CancelRecv()` as a receive operation cancellation function.

The `RWDPConn` class is used when sending WAP datagrams. This is not typically done; most WAP content is sent using WSP sessions.

Implementing the WSP layer

The WSP layer is implemented via two different classes. Like sockets at the TCP layers, WSP sessions can be connectionless or connected. A connectionless WSP session is suitable for the delivery of content requests and for the receipt of a single response. A connected WSP session is suitable for request and delivery over an error-free, reliable connection. These session types are analogous to connectionless and connected sockets. The biggest difference is that both WSP connection types must be made to accommodate both the push and pull models of data delivery.

Connectionless WSP sessions A connectionless WSP session is implemented by the `RWSPCLConn` class. I will break this class definition up a bit as we describe the class.

The `Open()` function is used to open a connectionless WSP session:

```
IMPORT_C TInt Open(RWAPServ &aServer,
      const TDesC8 &aRemoteHost,
      TPort aRemotePort,
      TPort aLocalPort,
      TBearer aBearer,
      TBool aSecureConn);
IMPORT_C TInt Open(RWAPServ &aServer, TPort aLocalPort);
```

As with the `RWDPConn` class, there are two versions of `Open()`. The first fully specifies both endpoints of the connection; the second specifies only the local endpoint.

To use the pull approach to collecting content, we use the `UnitInvoke()` and `UnitWaitResults()` functions below:

```
IMPORT_C TInt UnitInvoke(TFunction aFunction,
      const TDesC &aURI,
      const TDesC8 &aBody,
      const TDesC8 &aHeaders,
      TTransID aID);
IMPORT_C void UnitWaitResult(TDes8 &aBody,
      TDes8 &aHeaders,
      TTransID &aID,
      TWSPStatus &aWSPStatus,
      TRequestStatus &aRequestStatus);
    IMPORT_C void CancelUnitWaitResult();

    IMPORT_C TInt UnitWaitResult(TDes8 &aBody,
      TDes8 &aHeaders,
      TTransID &aID,
      TWSPStatus &aWSPStatus);
```

These functions come in synchronous and asynchronous versions, with the asynchronous version accompanied by the appropriate cancellation function. The `UnitInvoke()` function works like a GET command for HTTP, i.e., it opens a connection to a remote WAP server and sends both WAP headers and an optional content body. In the above definition, the `aURI` parameter specifies the destination (as an URI), and the body and WAP headers are sent as strings. The return of the function will fill in the `aID` variable. The response from this initial request is received using the `UnitWaitResult()` function. This function returns the body and WAP headers from the remote server's response. The `aID` value must match the initial request's ID, and the status of the response is filled in when the data is valid.

To wait for data to be pushed to a device, an application should use a `UnitWaitPush()` function:

```
IMPORT_C void UnitWaitPush(TDes8 &aBody,
      TDes8 &aHeaders,
```

```
        TPushID &aID,
        TRequestStatus &aStatus);
IMPORT_C void CancelUnitWaitPush();
IMPORT_C TInt UnitWaitPush(TDes8 &aBody,
        TDes8 &aHeaders,
        TPushID &aID);
```

Again, this comes in two versions—synchronous and asynchronous—and a cancellation function is provided for the asynchronous version. When called, these functions wait for data to be pushed to the device. Upon their return, the body and WAP headers are filled in, along with an ID to identify the data.

There's More!

The definitions to the data classes we have just described are in the `\Symbian\6.1\Quartz\Epoc32\include` directory. Look up the definitions to classes like `TTransID`, `TWSPStatus`, and `TPushID` just to understand how the data is represented. These are found in the file `wapcli.h`. In addition to these, pay attention to the `TReturnCodes` enumerations as well as the multiple definitions that are defined inside this class.

Connected WSP sessions A connected WSP session is implemented by the `RWSPCOConn` class. As before, opening a session is done with the `Open()` function:

```
IMPORT_C TInt Open(RWAPServ &aServer,
        const TDesC8 &aRemoteHost,
        TPort aRemotePort,
        TPort aLocalPort,
        TBearer aBearer,
        TBool aSecureConn);
```

There is only one version of `Open()`—the fully specified version. Like connected sockets, connected WTP sessions maintain a reliable connection over multiple data exchanges. So a specific target endpoint is required in the call.

Once created with `Open()`, a WTP session is connected to and disconnected from its endpoint using the following functions:

```
IMPORT_C TInt Connect(const TDesC8 &aClientHeaders,
                      class CCapCodec* aCap = 0);
IMPORT_C TInt Disconnect();
```

The `Connect()` function includes headers that are sent to the remote WAP server to set up the connected session and an optional

set of "capabilities" through a codec. A codec is a software module that compresses data for transport and decompresses the data upon receipt (the name "codec" corresponds to "compressor/decompressor"). This `CCapCodec` structure includes properties that characterize the connection session, including the size of data packets, size of buffers for sending and receiving, protocol options, and the like.

The data exchange for a connected session is initiated by the following function:

```
IMPORT_C TInt CreateTransaction(TFunction aFunction,
      const TDesC &aURI,
      const TDesC8 &aHeaders,
      const TDesC8 &aBody,
      RWSPCOTrans &aTrans);
```

The parameters for this call are similar to those used with `UnitInvoke()` functions for connectionless sessions, but there is some additional data: a description of the type of request that is being made must be provided in the `aFunction` variable), and an `RWSPCOTrans` object is returned. This latter object allows much more control over a connected session than a mere ID would.

Rather than the simple send/receive sequence that comprises a connectionless session, a connected session is characterized by a series of *events*, milestones in the exchange of data between both sides. These events drive a connected session. The `GetEvent()` function obtains the next event for processing:

```
IMPORT_C TInt GetEvent(TEvent &aEvent,
      RWSPCOTrans &aTrans) const;
IMPORT_C void GetEvent(TEvent &aEvent,
      RWSPCOTrans &aTrans,
      TRequestStatus &aStatus) const;
IMPORT_C void CancelGetEvent();
```

As usual, this type of function comes in synchronous and asynchronous versions, with the appropriate cancellation function. Sometimes several events are pending; if this is the case, the number of events can be obtained through the following function:

```
IMPORT_C TInt GetNrOfEvents(TUint &aNrOfEvents) const;
```

Once an event has arrived that has data attached to it, that data can be retrieved through the following function:

```
IMPORT_C TInt GetSessionData(TDes8 &aData,
      TSessionDataType aType) const;
```

This data is returned in a generic buffer, and the type of the data is available as well.

Given the above functions, the usual sequence for processing a connected session is as follows:

- Open a session to a specific endpoint.
- Connect the session through to the endpoint, setting the properties as necessary.
- Initiate a transaction with WAP headers and message content.
- Process events as they occur, exchanging data where necessary. This is much like processing connection states in an active object and may involve sending more data to the remote server.
- Close the session when the exchange is complete.

Several other operations are possible on connected sessions. These include suspending and resuming sessions, switching session bearers, and obtaining information about session progress.

There's More Here

There are many definitions for connected WSP sessions that we did not elaborate on. Look up the definition to the `RWSPCOTrans` class. Also, the `TEvent` class and the `TBearer` class, while applicable to more than `RSPCOConn` objects, make a lot of sense in this context. I recommend that you review them here.

Other WAP layers

Between the WDP and WSP layers, two other layers exist as support infrastructure. As applications using WAP content rarely manipulate these layers, I will not elaborate on them. However, for completeness, they should be mentioned.

The WTP layer is implemented by the `RWTPConn` class. This layer ensures the reliability necessary to support the WSP layer. Because of this, the definition of the class includes functions to initiate and acknowledge data packets that comprise upper layer sessions and functions to discover errors and gracefully recover from them. In addition, this layer has the only `Read()` and `Write()` in the stack implementation, indicating that it is in charge of the basic I/O operations for sessions.

The WTLS layer, sitting between the datagram delivery layers and the transaction layer, implements secure data encryption and provides an interface for managing secure connections. This layer is

implemented by the `RWTLS` class. To maintain secure connections, this layer implements functions that engage in negotiation with the remote side to provide authentication and to exchange encryption keys and algorithm specifications. It uses certificates (implemented by the `RCertificate` class) to verify the identity and keys of remote servers. It then encrypts each transaction that goes through the WAP session server. Note that the WAP session server has a `RWTLS` class that it uses in its operation. This can be null if no security is desired.

Certificates

Certificates are used by the WAP security layer as a way to identify remote servers and to obtain their encryption keys. A certificate is a digital document that "vouches for" the computer system. It guarantees that the server is what it says it is and ensures that it is not an imposter trying to spoof its way in to hack at security. Certificates are typically issued by servers run by third parties that are assumed to be truthful and secure themselves. Certificates are issued by these third parties and maintained by each individual device or server. Certificates can be verified by contacting the issuer.

The class that implements certificate information is the `RCertificate` class. You can find this definition in the file `\Symbian\6.1\Quartz\Epoc32\include\wapcli.h`.

16.2.3 Viewing WAP Content

Once the WAP content has been obtained, it must still be displayed. Like HTML, Symbian OS provides no architectural support for WML content viewing. Version 6.1 comes with a WAP viewer, and there is WAP engine support, but there is no support for an embedded viewing of WML contents. An application is responsible for displaying the WAP contents and abiding by WAP properties.

16.3 Summary

This chapter has overviewed the support Symbian OS gives to browsable content technologies, specifically HTTP/HTML and WAP. HTTP is considered an application protocol that is supported by TCP/IP; as such, it is not supported in a specific way by Symbian OS. WAP, however, is a new technology with its own protocol

implementations, and Symbian OS includes a complete implementation of the WAP stack. In both content situations, viewing the content is left to the application to implement; no support, such as embeddable viewers, is implemented in Symbian OS.

This chapter concludes the programming section of this book. In this past section, we have looked at how to communicate using Symbian devices over IrDA, through sockets, over Bluetooth and TCP/IP, through telephony and messaging technologies. We have also seen how Symbian OS supports Web (via HTTP and HTML) and WAP (through the WAP stack and WML) technologies. That's a wide range of possibilities for communication!

We have three more chapters to go, contained in the "Miscellaneous Topics" section. We will begin in the next chapter by examining the needs and support for device and PC synchronization.

Section 3

Miscellaneous Topics

17. Synchronization: PLP and SyncML pages 349–360
18. The Communications Database pages 361–377
19. Looking Ahead pages 379–386

17

Synchronization: PLP and SyncML

In Webster's dictionary, one of the definitions given for the word *synchronize* is "to ... arrange (events) to indicate coincidence or coexistence". This definition is very appropriate for the material in this chapter. The synchronization of devices is the process that arranges data objects to "indicate their coincidence" on both devices. Implementation of device synchronization is a black box to most people. We know the precondition for synchronization (i.e., that files and information might not be coincident) and the postcondition for synchronization (i.e., that files and information are indeed coincident). What happens between those two conditions has been somewhat of a mystery.

This chapter attempts to unravel this mystery to some extent. We will look at synchronization in general, and then concentrate on how the Symbian OS does synchronization. We will look at Psion Link Protocol (PLP) as the protocol currently used for synchronization, and we will examine a new method of synchronizing devices in SyncML. PLP represents Symbian's entry in the collection of proprietary protocols that are used for this task; SyncML represents the first attempt to standardize protocols for synchronization across devices and operating system manufacturers.

We will overview synchronization mechanics first. Once we understand the issues, we will look at some of the attempts in the industry and by Symbian to implement synchronization. We will then finish the chapter by focusing on PLP and SyncML.

17.1 Synchronization Mechanics

17.1.1 The Basics

The basic process of synchronization is easy to understand. Let's itemize this process:

- *Configuration phase:* Before we synchronize devices, we need to define *what gets synchronized.* Referring to our Webster's definition, we need to know *what data* have to be arranged and coincident between the devices. A typical collection of information that is synchronized might be one that consists of calendar information, to-do list items, email and other messages, and contact information.
- *Synchronization phase:* This phase is the one that is repeated constantly to make sure devices are in sync. Two devices are connected somehow, and applications are started on each one to facilitate the trading of data according to the configuration settings. Once the data objects have been arranged so that they are coincident, the two devices can be disconnected.

There are some aspects of synchronization that make it a unique process, distinguished from other data exchange processes (e.g., simple file transfer). First, synchronization is typically *automatic,* using the preset configuration to drive the exchange of data. Whenever the process is started, the same preconfigured series of steps is executed. Second, it often involves data format conversion. For example, to synchronize Agenda data on a Symbian OS device with a calendar application on a desktop computer, the Agenda data must be converted to a file of known format, say, Microsoft Outlook's format for calendar entries. That file must then be transferred to the desktop computer and converted into the format of the calendar application. Only then can each calendar entry be examined to see if it needs to be added to the calendar data set. Finally, in a synchronization exchange between two devices, both devices act as server and client. Requests for data are made by both sides, and both sides service data requests and send information back to the requestor.

17.1.2 Data Objects, Files, and Timestamps

Data objects are the items that are being transferred between the devices to be made coincident. It cannot be assumed that these items exist in a form that can be compared at any time. As stated above, file conversion may have to take place before information can be synchronized.

Note that *information* is synchronized, but *files* are the way that information is exchanged. The issue here is *granularity,* which is what the level of detail of synchronization is called. While files are exchanged (relatively coarse synchronization granularity), it is the individual entries within the files with which we are concerned (fine granularity).

One of the ways either side knows to synchronize information is by its *timestamp*. Differing timestamps in files are telltale signs that those files need to be synchronized. This is not, however, a simple copying of one file to replace another. Entries in each file are typically compared, and the information the files represent is merged together. That merged information is then sent to both devices as files to be installed as the new, synchronized information.

17.1.3 Communication Media

Any form of communication media that can be used to transfer data can be used to synchronize devices. As long as the medium enables bi-directional exchange, it is a usable medium.

The choice of synchronization media has a history. Early attempts at synchronization used serial cabled communication as a medium. This usage was expanded to include serial IR communication because the IR medium still worked as a serial medium. As communication, between devices was standardized, synchronization processes were also modularized to run over any communication medium that the two devices involved mutually supported. For example, many devices synchronize over USB connections using the same synchronization protocol implementations as they would for serial or network connections.

For most of their history, Symbian OS devices have used serial media. Most still do. As we will see, with the support developed for SyncML, the possibilities for synchronization media have expanded greatly.

17.1.4 Protocol Requirements

Many exchange protocols have been developed over the history of synchronizing devices. As these protocols have been designed, there has been a set of requirements developed that they try to satisfy.

The protocol must implement remote file handling. A synchronization client will need to browse the filesystem of the server. It may want to open files *on the server* and make the server read the contents, bringing data back to the client.

Implementations usually need to go beyond the mere copying of files. Extended functionality can include remote command execution and software package installation. It typically includes conditional copying or installation, based on timestamp comparison of source and destination objects.

The protocol must implement reliability and robust error recovery. This means that the protocol will likely be specified in layers,

like most of the protocols we have seen. These layers implement a modular design, with each focusing on a different aspect of file transfer, new functionality, or robustness.

Finally, the protocol will need multiple implementations. One will usually exist in the basic protocols supported by the local operating system. Another implementation is needed for the synchronization destination, e.g., a desktop computer. Sometimes multiple implementations are required for multiple desktop operating systems. If a software development version is needed for testing purposes, another implementation is usually included in the SDK.

17.2 Some Early Attempts at Synchronization

There have been many synchronization protocols. Each has its own niche and application.

File transfer protocols can probably be considered to be the first attempts at synchronization systems. While early protocols like XMODEM were simply designed to move files, applications developed around other protocols implement rudimentary synchronization. For example, some applications that implement the File Transfer Protocol (FTP) for the TCP/IP suite have facilities for checking timestamps and conditionally transferring files based on time or previously existing files. Backup systems were early attempts at synchronization; while the file transfer was one-way, backups made a crude synchronization system work.

Early in its work with handheld devices, Psion saw the need for some kind of synchronization protocol. Early computers (for example, the Organiser II, which was made in 1984) used the popular implementations for exchange protocols (e.g., XMODEM and YMODEM) at 9600 baud. This software was for file transfer only. On the Series 3, a new protocol debuted—an early version of PLP—and, through various machines, matured into the PLP protocol we have today. Symbian adopted this protocol from Psion.

The software used to drive synchronization has undergone much change. In the early days desktop computers had implementations of XMODEM or YMODEM built into terminal emulation programs. When Psion included PLP in the Series 3, it provided some early synchronization applications. RCOM was the first software to implement synchronization; this application was a command line interpreter that could take commands from a file (and thus allowed scripting of file synchronization). This evolved into an application with a GUI called PsiWin. Psion then implemented the software

driver side into a developer's kit and called it EPOC Connect. PsiWin became the version of EPOC Connect "badged" by Psion. Symbian took over the development of this software and renamed it Symbian Connect.

17.3 PLP and Symbian Connect

The current state of the art in synchronization on Symbian OS devices is the combination of PLP and the Symbian Connect software. In this section, we will overview the implementation and support that the Symbian OS gives to this pair.

17.3.1 The Psion Link Protocol

We gave a brief overview of PLP in Chapter 4; see Figure 4.3. PLP is specifically designed to use either the serial or the IR port. Its serial configuration uses hardware handshaking and a frame format of 8 bits, with 1 stop bit and no parity.

On top of the physical layer, PLP has a three-layer design. The link layer implements reliable transport of data, implementing error recovery via the retransmission of data frames. The Network Control Protocol (NCP) layer implements data stream multiplexing over this link layer. The NCP layer also implements flow control over each service's data stream. The Services layer implements the various services used by the remote device: software management, file manipulation and synchronization of files.

PLP allows the remote device to manipulate files on the local device and manage software installation. The Services layer implements the functionality needed to perform synchronization tasks. There are several servers that run at this layer to field and answer requests from clients.

- The *Link* server is started first and is used to communicate with the NCP layer. It starts the other servers as needed.
- The *Remote Protocol Command Service (RPCS)* server answers commands sent from the remote device. It supports general operations, such as launching and terminating processes. On some early devices, the `RPCS` server was not present by default; it was added by synchronization software upon usage.
- The *Remote File Services (RFSV)* server provides a remote interface that behaves similarly to local file accesses; files must be opened before they can be read or written, and a sequential file

pointer is maintained for each open file. A current directory is also maintained, allowing use of relative file names.

- The *Generic Custom Service (GENCSERV)* server is used by some designers to implement specialized services. For example, Ericsson implements a GENCSERV server that provides access to a device's contacts database. This is currently used on the Ericsson R380 phone, the first "one-box" phone.

It is the purpose of the NCP layer to multiplex data streams to and from each of these servers. Incoming data streams are first directed to the Link server, which makes sure the correct server is registered and running. Services layer packets include the name of the server they need to access; names for the servers differ slightly depending on the version of the OS and the model of the device.

17.3.2 Using PLP with Symbian Connect

A Symbian OS device will synchronize with a synchronization server; this server is typically a desktop computer running Microsoft Windows. The desktop computer must run software that can use the link protocol and drive the synchronization process; Symbian provides Symbian Connect for this purpose.

Through Symbian Connect, a software structure like that in Figure 17.1 is built. On the right side, data flows from the Symbian device—through its own PLP connection engine—to a server on the remote computer. This server sets up a connection and passes data to the connection engine, which is used by a desktop application

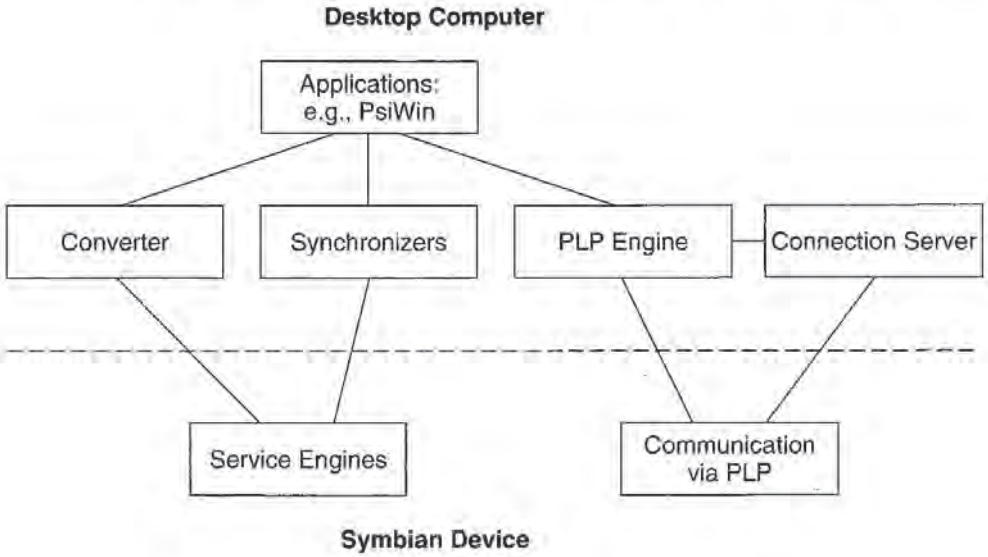

Figure 17.1 Symbian Connect synchronization structure.

to move data back and forth. When data moves between the two computers, synchronization services (using timestamps, file transfer, etc) and conversion services (converting files between computer application formats) are used.

This process requires that PLP clients and servers exist on both sides. On the Symbian OS device side, these functions are typically built into an application, typically called a "desktop link". On the desktop computer side, this is implemented by Symbian Connect.

The Symbian Connectivity Software Development Kit

To write applications that run on a desktop computer and use PLP to control a Symbian OS device, you need the Symbian Connectivity SDK. This is a collection of software, libraries, and source code examples that support the creation of software that can service and manipulate PLP. You can pick this up from Symbian's developer Web site.

Note that this SDK is only for computers that run Microsoft Windows.

17.4 SyncML

17.4.1 Proprietary is Problematic

Although PLP works well as a synchronization protocol, and is available for developers to use, it is a proprietary protocol. Its design is guarded information by Symbian, and official implementations exist only for Symbian OS devices and Symbian-OS-device-to-PC software. (Unofficial implementations are available for other operating systems; for example, Linux has an open source PLP implementation not sponsored by Symbian.)

Proprietary systems can restrict users. Indeed, while synchronization works well with Symbian Connect, Symbian OS devices are also restricted by Symbian Connect. For example, the only medium that is implemented for synchronization is serial communication; network synchronization would be a great benefit, as would synchronization with devices running other handheld operating systems. Along with Symbian, other manufacturers of other handheld devices also face this dilemma: each handheld operating system uses proprietary protocols for synchronization and is limited by the implementation of protocols by its manufacturer.

SyncML is an attempt to solve this problem. SyncML is the result of an initiative supported by a consortium of handheld device

hardware and software companies. This consortium sought to create a single, common synchronization protocol and format specification that all its members would support. Several hundred companies support this initiative, with Symbian as one of the charter sponsors of the initiative.

SyncML support is expected to be implemented by large numbers of the consortium members in their products in 2002.

17.4.2 SyncML Characteristics

The benefits of using a single synchronization platform for different devices are many. Using a single implementation of desktop computer software (which can be developed by third parties), devices from many vendors can synchronize their data.

SyncML's design is broad enough that devices are now limited by their own capabilities—not that of the synchronization protocol. A wide variety of media can be used: serial, Bluetooth, and Ethernet networks. New and interesting services can be implemented—imagine a "synchronization service provider" that can be used over the Internet.

The designers of SyncML had to grapple with several criteria:

- *The system must support current, popular data formats.* These include the standard vCard and vCalendar data formats that we have seen in this book, as well as collaborative formats such as email.
- *The system must support binary data formats.* Synchronization can include the exchange of executable code or databases—that is, data objects that do not have a neat textual representation.
- *The system must be extensible.* As new data formats are designed, the SyncML system must extend to accommodate these as well.
- *The system should build on existing transport technologies.* Current transport methods include serial communication and networking, as well as transport protocols like HTTP, SMTP, and WSP that ride on top of these other methods. The system should embrace all of these technologies.

The consensus in designing SyncML was that the system should not mandate data formats. It should be centered on format *description* rather than specific format *implementation*. The consensus was also that the data transport should work in layers that are as high as possible in a protocol stack—i.e., transport should not be restricted, but applications should embrace as many transport methods as possible.

To implement these criteria, SyncML is based on Extended Markup Language (XML). XML is a data description language that can be used to describe any kind of structured data stream and how application software (such as browsers) are to deal with it. It has been suggested that XML will replace HTML as the language for describing Web pages. This would be possible, because HTML is one of the basic structured streams that XML specifies and recognizes. Other examples might be database records, scientific data to be visualized in a browser, and (of course) synchronization data such as vCalendar and vCard format entries. Naturally, future data formats, which do not exist today, can be specified with XML as well.

Consider an example. Assume that we want to transfer a contact to a synchronization server. As part of a larger SyncML data stream, we might represent the contact data as a vCard and transfer that vCard as the SyncML format in Figure 17.2. Let's make a few observations about this example:

```
<Add>
    <CmdID>12345</CmdID>
    <Cred>
        <Meta>
            <Type xmlns="syncml:metinf">syncml:auth-md5</Type>
            <Format xmlns="syncml:metinf">b64</Format>
        </Meta>
        <Data>MJJkNDI2x7ZjNjgwMTNiYWZk455yN2JjMjN1ZDM4YzENCg==</Data>
    </Cred>
    <Meta>
         <Format xmlns="syncml:metinf">chr</Format>
         <Type xmlns="syncml:metinf">text/x-vcard</Type>
     </Meta>
     <Item>
         <Source>
            <LocURI>./card</LocURI>
         </Source>
         <Data>BEGIN:VCARD
          VERSION:2.1
          FN:Mike Jipping
          N:Jipping;Mike
          TEL;WORK;VOICE:+1-616-555-1234
          TEL;WORK;FAX:+1-616-555-4321
          EMAIL;INTERNET:mike@jipping.com
          END:VCARD
          </Data>
     </Item>
</Add>
```

Figure 17.2 Example SyncML vCard representation.

- Note that the format is full of bracketed tags, like HTML and its parent SGML. As in HTML, tags are used to delineate sections of

the specification. In the example, we have four main sections: `CmdID`, `Cred`, `Meta`, and `Item`.

- The `CmdID` gives the code of the function being used in the current operation (e.g., `Add`).
- The `Cred` section gives the authentication information. In this example, the credentials are given using an MD5 hash algorithm for encryption and base 64 encoding.
- The `Meta` section gives details about the data structuring. In this example, we are using character format (`chr`) with a type encoding of a vCard (`text/x-vcard`).
- The `Item` section gives the actual vCard specification. In this example, we have specified the source of the vCard (the `<Source>` tag) and then given the vCard data (the `<Data>` tag).

In addition to a data description language, SyncML specifies a common, universal data transfer protocol. The SyncML standard specifies that many protocols can transfer SyncML data. However, it also specifies a specific synchronization protocol that can tunnel through existing protocols. Supporters are also free to use their own proprietary protocols to support the data transfer if they wish.

Figure 17.3 holds a diagram of the protocol stack for SyncML. Synchronization applications communicate with each other through SyncML, via its data formatting and its universal data transfer protocol. The SyncML data format is then transmitted over some

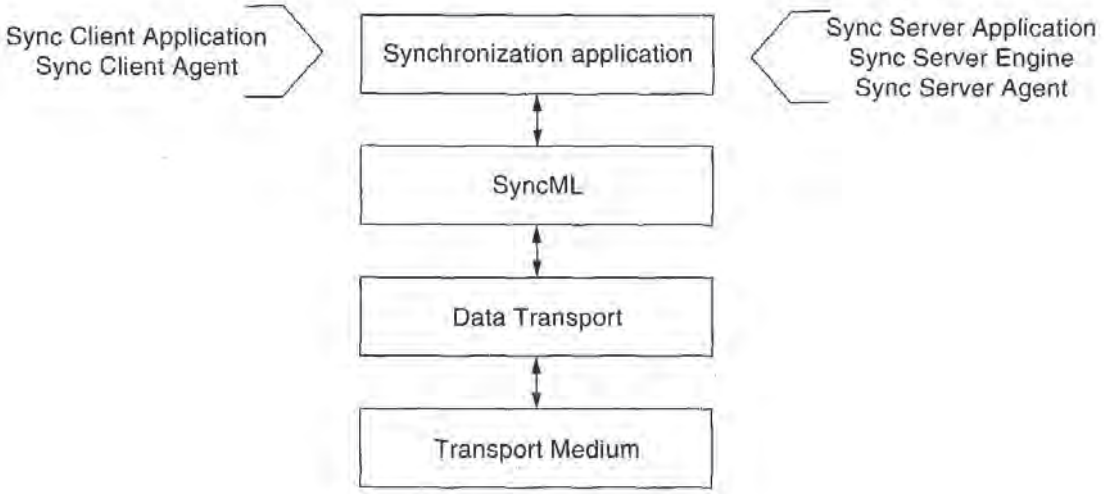

Figure 17.3 *SyncML protocol stack.*

transport protocol and medium to the remote device, which passes the received data through SyncML and incorporates the result.

Note that protocols may—in fact, most certainly will—encapsulate the SyncML data in other packages for transport. Certainly, data packets are wrapped by protocol data packets as they move through protocol layers. SyncML data will probably be formatted as MIME objects for use in email. There have also been suggestions that, on devices with restricted resources or limited bandwidth, SyncML be encoded in a binary format (e.g., WAP Binary XML (WBXML), a binary format for transmission over WAP protocols). This binary encoding would reduce the size of the data as well as the length of the transmission time.

More Information

More information about SyncML is available from Internet sources. The core site for SyncML is ***http://www.syncml.org***. In addition, the many manufacturers that back SyncML have information on their sites that encourage more learning and software development. Nokia, for example, has an extensive segment of their site dedicated to SyncML (see ***http://www.forum.nokia.com/main/1,6668,1`47,00.html***).

17.4.3 Support for SyncML

Specifications for SyncML are very new. Originally planned for late 2000, the specifications were officially published in mid-2001. Implementations have started surfacing during 2001. For example, Nokia has included SyncML support in some of its newest phones (e.g., the 9210 and 7650).

Recall that SyncML is both a data format and a data protocol. One would imagine that SyncML support would take the form of a socket and, as the data protocol is implemented, that is the form it will likely take. However, most of the effort for SyncML support has been concentrated on the data format, and this takes the form of library support for format conversion.

In version 6.1 of the Symbian OS, there is no support delivered with the distribution. However, support can be added later; for example, Nokia has delivered extensions to its emulators that upgrade them to support SyncML.

Upgrade Your Emulator

If you want to upgrade your emulator installations to run SyncML, check this book's support Web site for pointers to SyncML applications: Nokia's SyncML client for both the 9210 and 7650 emulators, and Symbian's upgrade for their reference emulator implementation.

17.5 Summary

In this chapter, we have highlighted device synchronization. We reviewed the issues and the history of synchronization. We dug into Symbian's current support of PLP and Symbian Connect as their current synchronization solution, and we overviewed SyncML as the next implementation of synchronization. SyncML represents the future of this area and will be supported in upcoming releases of the Symbian OS.

The next chapter looks in detail at the Symbian OS communications database which is used to store persistent communication settings for devices and applications.

18

The Communications Database

If one could "configure" a social engagement much like one can configure a computer system, going to a party would be a whole lot easier. If this were the case, one could simply set up a central area with personal details. Anyone entering the party would check this information center first and all the initial details about meeting someone would be taken care of straight away. While people would still scramble to figure out what to chat about, this would cut down on repeating your name and occupation for each new person you meet.

This is the idea behind the use of a communications database to centrally store all communication configurations for a Symbian OS device. The database stores those common items that must be repeated for each application that uses communication. For instance, each TCP/IP application must know about your ISP, and each SMS messaging application must know about the service center number you want to use. As the lifespan of this database extends beyond each application's execution time, it can be referenced for the common configuration items that it holds for all implementations in the communications architecture.

We will discuss this communications database in this chapter. We will start by reviewing some issues that configuration of communication involves. We will then spend the majority of the chapter reviewing the API for accessing and manipulating the communications database.

18.1 Configuration and the Communications Database

Configuration issues have dogged computer systems for a long time. Configuration settings must be persistent and must outlast the execution of any single application. At the same time, many

applications must be able to access configuration settings and alter them, hopefully through a common, efficient interface. Sometimes applications make changes that should be permanent; sometimes changes should be temporary and thus be discarded before an application terminates.

These aspects of configuration have resulted in different functions of storing and maintaining settings on different systems. The Unix operating system, for example, takes the lowest common denominator approach. Unix configuration settings are kept in files, stored on accessible disks, in readable textual format with no rules as to how to specify settings (i.e., there is no standard API). Humans and applications can read configuration files and alter them easily. The Unix approach specifies that since any application should be able to read text files, text storage is the best (i.e., simplest) function. Microsoft has taken a more organized, proprietary approach in Windows. The Windows registry is a hierarchical, highly organized way of keeping configuration settings. With the registry, an application works with settings from a category hierarchy (e.g., categories like `HKEY_LOCAL_MACHINE/SOFTWARE/Psion/PsiWin`) in key/value pairs. Microsoft provides an API for programmer access to this registry and tools for users and administrators to access it. Still other operating systems store crucial information in system-maintained tables, available only to applications that make system calls to retrieve this information from the OS kernel (e.g., older VAX systems running Open VMS).

After considering these different ways of storing information, the Symbian OS designers made several important design decisions about settings for communication resources. First, they decided that settings should be applied on a system-wide basis for all applications. Using this idea, a configuration editor could be used to affect all applications (e.g., the Control Panel on some Symbian OS devices). Second, they decided that configuration information should be visible and changeable by all applications. Finally, they decided to store configuration information in a database form, accessible through an API that uses a database context. This meant that they could implement it via included database mechanisms and that the API would be familiar to many Symbian OS programmers.

18.2 The Communications Database API

Symbian OS database implementation is based on a standard database management system (DBMS) implementation. While I will

explain a bit about DBMS concepts, I will not be defining them in detail. It is helpful here to be familiar with concepts such as database tables, views, and record selection. These comprise a large part of the DBMS implementation.

18.2.1 Database Structure

The communications database is structured as a standard Symbian OS relational database, managed through DBMS procedures. As a single database, it holds several data tables. Each data table is manipulated by constructing a view of the table and working with its records through that view. The tables contained in the communications database are listed in Table 18.1, along with the names used to access them. Tables without names (listed with N/A as the name in Table 18.1) are accessed using special function calls that we will discuss later.

Therefore, to access the communications database, you must know which table you need to access and how you want to access it. The standard way to access this structure is through the following steps:

- Create a new database object.
- Use this database object to create a view of a data table.
- Select a record—either by creating a new record or by finding an existing one.
- Manipulate this record—read from or write to it as needed.
- Commit any changes made to the record.
- When completed, delete the view and database objects.

18.2.2 Communications Database Classes

The communications database is implemented via the `CComms-Database` class. In turn, this class is derived from the `CComms-DatabaseBase` class. Together, these classes define a very large number of functions which I will summarize here.

Review the Communications Database Definition

Take some time to review the definitions setup for the communications database. In the `\Symbian\6.1\Quartz\Epoc32\Include` directory, pay specific attention to the `cdbcols.h` file for table-naming macros and to the `commsdb.h` file for class definitions.

Table 18.1 Communications database tables

Table	Table Name	Description
Incoming Connection Preferences	N/A	Order ranking for IAP settings for incoming connections
Outgoing Connection Preferences	N/A	Order ranking for IAP settings for outgoing connections
Global Settings	N/A	Miscellaneous settings that govern most connection configurations
Internet Access Point	IAP	The settings used to access the Internet (e.g., which modem)
Outgoing ISP	DialOutISP	Connection settings for the ISP used for outgoing messages
Incoming ISP	DialInISP	Connection settings for the ISP used for incoming messages
Outgoing GPRS Provider	OutgoingGPRS	Settings used for the GPRS provider for outgoing data
Incoming GPRS Provider	IncomingGPRS	Settings used for the GPRS provider for incoming data
Default GPRS Provider	DefaultGPRS	Default settings for a GPRS provider
Modem List	Modems	A list of modems available on the device and their settings
Location List	Locations	A list of phone settings for various locations
Charge Cards	ChargeCard	A list of charging settings used to make long distance phone calls
Proxy List	Proxies	A list of HTTP proxy servers and their settings
WAP Access Point List	WAPAccessPoint	A list of sites to access WAP data
WAP IP Bearers	WAPIPBearer	A subset of the WAP access point list that are IP bearers, including their IP settings
WAP SMS Bearers	WAPSMSBearer	A subset of the WAP access point list that are SMS bearers, including their settings
Bluetooth Device List	BTDeviceTable	A list of Bluetooth devices available
Bluetooth Default Settings	BTDefaultTable	The default settings used for a Bluetooth device
Bluetooth Security Settings	BTSecurityTable	Security settings for Bluetooth devices

Creating the database object

New instances of the `CCommsDatabase` class are created with the `NewL()` functions, defined below:

```
IMPORT_C static CCommsDatabase* NewL(
   TCommDbDatabaseType aType);
```

```
IMPORT_C static CCommsDatabase* NewL(
   TCommDbDatabaseType aType,
   TCommDbOpeningFunction &aOpeningFunction);
```

These two functions must be given the type of database that is to be created. `TCommDbDatabaseType` is an enumeration, giving three types of databases: an Internet Access Provider database, an Internet Service Provider database, and a generic, unspecified type of database. An IAP database is identical to an ISP database, except that the former provides information about providers of Internet access, and the latter does not include these. However, these distinctions are moot in version 6.1 of Symbian OS, because the ISP database has merged into the IAP database. The second function also requires a specification of the way the system is to open the database: creating it, copying it, or opening an existing one.

Creating a view of the database

Once a database object has been created, the next step is to create a view of a data table. A table view is encapsulated by the `CCommsDbTableView` class. We can get an instance of this class in several ways. The easiest way is to get it using a `CCommsDatabase` object, using one of the functions defined below:

```
IMPORT_C CCommsDbTableView* OpenTableLC (
   const TDesC &aTableName);
IMPORT_C CCommsDbTableView* OpenViewLC (
   const TDesC &aTableName, const TDesC &aSqlQuery);
```

The first function takes the name of the table we need to open and creates a view that makes all records in the table accessible. The name of the table is one from Table 18.1 (which can be found in the constants in the `cdbcols.h` file). The second function uses the name of the table to open an SQL query to select specific records out of the table that we are opening. Structured Query Language (SQL) is a standard DBMS manipulation language and is the language that Symbian OS uses to manage its database systems.

As an example, let's say that we want to list all the charge cards that are stored in the communications database. We would open the database and create a view of the `ChargeCard` table that would give us all table entries. We could do this in the following way:

```
void CDB::ViewEntries()
{
   CCommsDatabase *db;
   CCommsDbTableView *view;
```

```
    TInt result, count;
    TBuf<40> Name, AcctNumber, Pin;
    // Open the database and get a table
    db = CCommsDatabase::NewL(EDatabaseTypeIAP);
    view = db->OpenTableLC(TPtrC(CHARGECARD));

    // ... more code to view all entries

    delete view;
    delete db;
}
```

We will fill in the commented-out portion shortly. Using a view created with this code, the program could display the output shown in Figure 18.1.

Figure 18.1 Example listing of the ChargeCard table.

Now let's say that we wanted to only view entries with non-zero PINs. We could substitute the `OpenTableLC()` call with a `OpenViewLC()` call, which requires the use of SQL. This function needs an SQL view query, which is generated by an SQL `SELECT` statement. This statement has the following general form:

```
SELECT  selection-spec FROM  table-name
                [WHERE  condition] [ORDER BY  sort-spec]
```

The *selection-spec* field is either a list of record fields or a "*" to select all fields. The *table-name* is the name of the table with which we are concerned. The *condition* is a Boolean condition that will be used to select specific records. The *sort-spec* is a list of fields that are used as sorting keys on which to sort the resulting records. For example, to view all entries except those with PINs equal to "0000", we could replace the `OpenTableLC()` call with the function call below:

```
view = db->OpenViewLC(TPtrC(CHARGECARD),
          _L("SELECT * FROM ChargeCard WHERE Pin <> '0000'"));
```

In this code, we specify the `ChargeCard` table, but we add an SQL statement to view only the entries we want. The result of using this to define our view is in Figure 18.2.

Figure 18.2 Example listing of ChargeCard table records whose PIN () 0000.

I should make two points about this use of an SQL statement:

- Notice we give the table name *twice*: once in the first parameter and once in the SQL statement. This is unavoidable; leaving either one out causes a system panic.
- Those who are used to writing SQL for relational database queries will want to terminate this SQL statement with a semi-colon (this is standard syntax). However, this statement *should*

not have a terminating semicolon; inserting one will cause a system panic.

Using SQL in this way is the most flexible way to select and order database table entries. If you do not want to wrestle with SQL statements, however, there are more ways to open a view and select records. These functions still use SQL to query the communications database, but they do not require the programmer to understand SQL, just to be able to use a simple API.

```
IMPORT_C CCommsDbTableView* OpenViewMatchingUintLC(
   const TDesC &aTableName, const TDesC &aColumnToMatch,
   TUint32 aValueToMatch);
IMPORT_C CCommsDbTableView* OpenViewMatchingBoolLC(
   const TDesC &aTableName,const TDesC &aColumnToMatch,
   TBool aValueToMatch);
IMPORT_C CCommsDbTableView* OpenViewMatchingTextLC(
   const TDesC &aTableName, const TDesC &aColumnToMatch,
   const TDesC8 &aValueToMatch);
IMPORT_C CCommsDbTableView* OpenViewMatchingTextLC(
   const TDesC &aTableName, const TDesC &aColumnToMatch,
   const TDesC16 &aValueToMatch);
```

Each of these functions requires a table name as the first parameter. The second parameter is the name of the column that we want to perform the selection over. The third parameter is the value to match—an integer, Boolean, or string (8-bit ASCII or 16-bit Unicode). Note that each function performs a straight equality match. Unfortunately, this means that a selection like that in our example above would not be possible with these functions; they cannot specify inequality (e.g., "`...WHERE PIN <> '0000'`").

Using SQL for Selections

If you know SQL, it is by far the best way to select records from the communications (or any Symbian OS) database. It is the most flexible query mechanism and can streamline a search for items. The SQL used by the Symbian OS DBMS has a few small differences from conventional SQL. I have already highlighted differences in semicolon usage, and wildcards in the WHERE clause use "*" instead of "%". Using SQL for searching minimizes the coding effort for the search process; you can find records with SQL for which you would have to implement many lines of code in an application without it.

SQL is worth learning if you do not know it already. Two books worth reading to learn SQL are *Beginning SQL Programming* by

John Kaufman, Brian Matsik, and Kevin Spencer, published by Wrox Press, and *The Practical SQL Handbook* by Judith Bowman, Sandra Emerson, and Marcy Darnovsky, published by Addison Wesley.

18.2.3 Accessing and Reading Records

Once we have established a view as an object of the `CCommsDbTableView` class, we need to find the records we want to manipulate. Three functions are provided to navigate through a view's records. They are defined below:

```
IMPORT_C TInt GotoFirstRecord();
IMPORT_C TInt GotoNextRecord();
IMPORT_C TInt GotoPreviousRecord();
```

These allow us to find the first record in a view, then to move to the next or previous record based on a current record position. Should any of these functions find that it cannot navigate as requested, it will return the `KErrNotFound` error code. The database system maintains the concept of a "current record position". All record operations are performed on the record at the current record position. Note that there is no specific class that represents a communications database record.

Record data is read column-by-column, using one of the `CCommsDbTableView` functions below:

```
IMPORT_C void ReadUintL(const TDesC &aColumn,
   TUint32 &aValue);
IMPORT_C void ReadBoolL(const TDesC &aColumn,
   TBool &aValue);
IMPORT_C void ReadTextL(const TDesC &aColumn,
   TDes8 &aValue);
IMPORT_C void ReadTextL(const TDesC &aColumn,
   TDes16 &aValue);
IMPORT_C HBufC* ReadLongTextLC(const TDesC &aColumn);
```

These functions read integer, Boolean or string data, from the column whose name is given as the first parameter, into the variable given as the second. For string reads, there are three versions: one that reads data as 8-bit ASCII values, one that reads data as 16-bit Unicode values; and one that will read a value of any length into a string. While the variables in the `ReadTextL()` functions are bounded in size (they are preallocated), the `ReadLongTextLC()` function reads a column value into the heap and returns a pointer

to where the results are stored. In the latter case, the function takes care of string allocation.

Now we can complete our example that prints `ChargeCard` table entries. The complete function definition is given below:

```
void CDB::ViewEntries()
{
   CCommsDatabase *db;
   CCommsDbTableView *view;
   TInt result, count;
   TBuf<40> Name, AcctNumber, Pin;


   // Open the database and get a table
   db = CCommsDatabase::NewL(EDatabaseTypeIAP);
   view = db->OpenTableLC(TPtrC(CHARGECARD));


   // Walk through each record, retrieve fields and print
   count = 1;
   result = view->GotoFirstRecord();
   while (result == KErrNone) {
      view->ReadTextL(TPtrC(COMMDB_NAME), Name);
      view->ReadTextL(TPtrC(CHARGECARD_ACCOUNT_NUMBER),
                      AcctNumber);
      view->ReadTextL(TPtrC(CHARGECARD_PIN), Pin);
      console->Printf(_L("Entry #%d:\n %S\n %S\n PIN: %S\n\n"),
         count++, &Name, &AcctNumber, &Pin);
      result = view->GotoNextRecord();
   }


   delete view;
   delete db;
}
```

In the code above, note that I use standard defined macros for the names of the fields in the `ChargeCard` database.

18.2.4 Creating and Editing Records

New database records are inserted at the current record position through a call to the `CCommsDbTableView` function defined below:

```
TInt InsertRecord(TUint32 &aId);
```

This function will make all preparations for inserting a record and make that record the one at the current record position. The ID of this new record is returned in the parameter.

To write into entries, we first need to indicate our intentions. For new records, we can use the `InsertRecord()` function defined above. For existing records, we need the function defined below:

```
TInt InsertRecord(TUint32 &aId);
TInt UpdateRecord();
```

A panic will be raised if any writes are made to a database record without first calling `InsertRecord()` or `UpdateRecord()`. One of these must be called for each set of changes made to an existing record.

To actually modify a record, we need to use one of the functions defined below:

```
IMPORT_C void WriteUintL(const TDesC &aColumn,
   const TUint32 &aValue);
IMPORT_C void WriteBoolL(const TDesC &aColumn,
   const TBool &aValue);
IMPORT_C void WriteTextL(const TDesC &aColumn,
   const TDesC8 &aValue);
IMPORT_C void WriteTextL(const TDesC &aColumn,
   const TDesC16 &aValue);
IMPORT_C void WriteLongTextL(const TDesC &aColumn,
   const TDesC &aValue);
IMPORT_C void SetNullL(const TDesC &aColumn);
```

The forms here are analogous to those used to read from entries, in that writing is done by entering specific values into specific columns. The name of the column is given as a string for the first parameter and the value to be written is given as the second. As with reading, we can write using three data types: integer, Boolean, and string. Strings can be written in 8-bit and 16-bit forms, and long, unlimited strings allocated from the heap may be written using the `WriteLongTextL()` function. A specific column may be made to be null (i.e., having *no* value, as opposed to a blank value) by the `SetNullL()` function.

When changes to a record have been completed, these changes need to be committed to the database. Commitment of changes is done on a record-by-record basis; we cannot commit changes to several records at once. We commit changes by calling the function defined below:

```
IMPORT_C TInt PutRecordChanges(TBool aHidden = EFalse,
   TBool aReadOnly = EFalse);
```

This call allows some tailoring to be done to the record as it is committed to the database. The record can be designated as hidden; it can also be entered as read-only. These attributes are

retained until the next commitment of changes to it in the database. Prior to closing access to the database, changes to a record can be cancelled by using the following function:

```
void CancelRecordChanges();
```

This is effective for both new (started with `InsertRecord()`) and existing records (started with `UpdateRecord()`).

As an example, let's say we continue our crusade against zero PINs. We want to change all PINs in the `ChargeCard` table that are 0000 to have the value '4321'. We could use the code fragment below to do this:

```
view = db->OpenViewLC(TPtrC(CHARGECARD),
_L("SELECT * FROM ChargeCard WHERE Pin = '0000'"));
result = view->GotoFirstRecord();
while (result == KErrNone) {
   view->UpdateRecord();
   view->WriteTextL(TPtrC(CHARGECARD_PIN), _L("4321"));
   view->PutRecordChanges();
   result = view->GotoNextRecord();
}
```

18.2.5 Using Transactions

There are times when we should be concerned that allowing multiple users to simultaneously access the communications database might cause problems. For example, if a process is reading the entries that another process is changing, the reader might get the wrong information. Transactions help to avoid these situations.

A *transaction* is a set of statements that is guaranteed to run *atomically*. This means that a transaction will either run from start to finish with no interference from other processes or it will not be run at all. This is useful for both database readers and writers; readers can be assured of no changes during a read, and writers can be assured that the written data will be used as a unit with no half-completed writes conveying bad information.

The `CCommsDatabaseBase` class provides functions to implement transactions. A transaction is bracketed by the functions defined below:

```
IMPORT_C TInt BeginTransaction();
IMPORT_C TInt CommitTransaction();
```

Any statement between the two functions above is assumed to be part of the transaction. The system implements the transaction as follows:

- The use of the `BeginTransaction()` function causes the system to try to obtain a *read-lock* on the database. This type of lock is shared by readers and, while only read operations are being used, multiple read-locks may coexist on the database.
- At the time the first write operation is used, the system converts the read-lock obtained by the process to a *write-lock*. Write-locks are exclusive, and no read- or write-locks may coexist with a write-lock.
- When the `CommitTransaction()` function is used, the lock held by the process committing the transaction is removed, and any changes made during the transaction are committed to the database.

If a write-lock is already held by another user when a `BeginTransaction()` call tries to obtain a read-lock, the function call will return a `KErrLocked` error code. Note that changes to the database made during a transaction still require additional `PutRecordChanges()` calls; the `CommitTransaction()` function will not take care of operation commitment.

It is possible to *rollback* changes—that is, reverse them—if a transaction is terminated with the following function:

```
IMPORT_C void RollbackTransaction();
```

This function call is used in place of calling `CommitTransaction()`. It causes all changes to the database to be discarded and removes any locks.

Information about transactions and locking status can obtained by the following functions:

```
IMPORT_C TBool InTransaction();
IMPORT_C TBool IsDatabaseWriteLockedL();
```

If a transaction is in progress—determined by whether a process has made a `BeginTransaction()` call—then the `InTransaction()` function will return an `ETrue` value. If the database has been write-locked by a process, the `IsDatabaseWriteLockedL()` will return an `ETrue` value.

Let's rework our anti-zero-PIN example one more time. This time, we will change the way we select records, and we will group all changes in a transaction. The code is below:

```
view = db->OpenViewMatchingTextLC(TPtrC(CHARGECARD),
    _L("Pin"), _L("0000"));
db->BeginTransaction();
result = view->GotoFirstRecord();
while (result == KErrNone) {
```

```
    view->UpdateRecord();
    view->WriteTextL(TPtrC(CHARGECARD_PIN), _L("4321"));
    view->PutRecordChanges();
    result = view->GotoNextRecord();
  }
db->CommitTransaction();
```

During this change to the `ChargeCard` table, no readers or writers can access the database.

18.2.6 Special Purpose Tables

Table 18.1 lists the tables in the communications database. While you are encouraged to explore all of them, and most are self-explanatory, there are a few that need further explanation.

The connection preferences tables contain information about the ordering of Internet Access Points (IAPs) in a connection sequence. These IAPs are tried in order until a connection succeeds. There is a sequence specified for incoming connections and for outgoing connections. The information in the table contains a ranking of the IAP in the sequence, the type of bearer that should be used, a determination of how to interact with the user about this IAP (e.g., conversing with the user through a dialog before connecting or connecting automatically), and the direction of the connection. Connection preference tables are represented by their own class (the `CCommsDbConnectionPrefTableView` class); this class has its own functions to open and view each table as well as to browse each table in a specific order.

There is a special table that holds "global" settings. These are settings that govern all connection attempts. They are miscellaneous settings like the number of dial attempts and whether a device accepts incoming GPRS connections. To use these settings, no view needs to be established; a table is opened automatically when the communications database object is created. Global settings can be read, written, and cleared using the function definitions below:

```
IMPORT_C void GetGlobalSettingL(const TDesC &aSetting,
   TUint32 &aValue);
IMPORT_C  void GetGlobalSettingL(const  TDesC &aSetting,
   TDes &aValue);
IMPORT_C void SetGlobalSettingL(const TDesC &aSetting,
   TUint32 aValue);
IMPORT_C  void SetGlobalSettingL(const TDesC  &aSetting,
   const TDesC &aValue);
IMPORT_C void ClearGlobalSettingL(const TDesC &aSetting);
```

Settings have either integer or string values. Each function takes the name of the setting to access as the first parameter. `GetGlobalSettingL()` fills the second parameter; `SetGlobalSettingL()` takes the new value from the second parameter.

Examine the Global Settings

The `cdbcols.h` file has the set of global settings that can be used. In addition, browsing the communications database via the `setupcomms` shell command will show you all the current settings of these global parameters.

18.2.7 Other Functionality

In addition to the large core of operations that we have covered, there are a few more concepts that we need to deal with before we leave communications databases. We will overview them here.

Deleting records

No set of operations would be complete without the ability to delete. Deleting records is accomplished by the following function:

```
IMPORT_C TInt DeleteRecord();
```

As with other database changes, deleting a record must be committed to the database by calling `PutRecordChanges()`.

Notification of changes

Applications can be notified by the system when changes are made to the communications database. Notification involves waiting for the system to send a signal, and this kind of situation naturally calls for an active object. As with other active object scenarios, notification of database events requires that an active object makes a request and uses `SetActive()` to return control to the operating system. This notification request is made via the `CCommsDatabaseBase` function below:

```
IMPORT_C TInt RequestNotification(TRequestStatus &aStatus);
```

As is the standard operating procedure, this asynchronous call includes a status variable through which we can receive database update information. This request can be cancelled through the function below:

```
IMPORT_C void CancelRequestNotification();
```

Upon notification of a database event, the type of change that occurred is denoted in the value of the status variable passed the request function. The value of this variable is defined by the enumerators of the `RDbNotifier::TEvent` structure.

Hidden records

Records can be designated as hidden in the communications database. Hidden records do not show up in normal record selections. To allow hidden records to be selected, an application must call the `CCommsDatabaseBase` function below:

```
IMPORT_C void ShowHiddenRecords();
```

Records are designated as hidden when the `PutRecordChanges()` function is called. Including the first parameter with an ETrue value will make this designation.

Specialized functions

There are a few functions in the class collection for the communications database that are not specifically tied to tables or setting, but are useful nonetheless.

- The functions below resolve phone numbers:

```
IMPORT_C static void ResolvePhoneNumberL(
   TDesC &aNumber, TDes &aDialString,
   TParseMode aDialParseMode,
TUint32 aLocationId, TUint32 aChargecardId);
IMPORT_C void ResolvePhoneNumberFromDatabaseL(
   TDesC &aNumber, TDes &aDialString,
   TParseMode aDialParseMode,
TUint32 aLocationId, TUint32 aChargecardId);
```

These functions from the `CCommsDatabaseBase` class will take a phone number in the first parameter and translate it into the second parameter, using the specific parsing function, dialing location and charge card given in the rest of the parameters.

- Each table can have a *template record*, a record that is copied to a new record whenever one is created. This is useful for establishing default values in records. Template records are implemented by their own class (the `CCommsDbTemplateRecord` class) and are manipulated in the same way standard communications database records are manipulated.
- The communications database uses the idea of *override settings*. These settings are *(tablename, columnname, value)* triples that

override the settings in the communications database for a single connection only. These settings are implemented by their own class (the `CCommDbOverrideSettings` class). The use of these settings provides a way to change a single configuration set without changing the database.

18.3 Summary

This chapter has dealt with the Symbian OS communications database, the database which holds the configuration settings for Symbian OS communication systems. These settings are to be persistent between application executions and can be changed by applications. We discussed configuration issues and spent most of the chapter reviewing the API for manipulating the communications database.

The next chapter is the final chapter in this book. In it, we discuss the future of communications support on Symbian OS.

19

Looking Ahead

The computer field is strewn with empty predictions by intelligent people. Thomas Watson, when he was chairman of IBM, said in 1945 that "there is a world market for maybe five computers". Ken Olsen, then the president and founder of Digital Equipment Corporation, said in 1977 "There is no reason why anyone would want a computer in the home". Even Bill Gates, founder of Microsoft, said about a computer's memory capacity, "640 K should be enough for anybody". He said that in 1981.

Nevertheless, I am going to finish this book by looking into the future and making a few predictions. We have spent this book looking at Symbian OS version 6.1. At this writing (January 2002), Symbian OS version 7.0 is being launched. These new development efforts will undoubtedly test the communication architecture by implementing new ways to use communication technology. New media, new protocols, and new forms of content will likely be incorporated in hardware and software for Symbian OS devices.

This chapter looks at what is possible for future implementation. There are some obvious technologies; we will see these first. Then there are some technologies way out there on the horizon. We will mention these in closing.

19.1 The Next Step: GPRS

Symbian OS version 7.0 contains support for GPRS integrated into the messaging infrastructure. There is some support in version 6.1.

19.1.1 What is GPRS?

General Packet Radio Service (GPRS) is a bridge protocol, implemented for 2.5G networks. Second generation (2G) networks are

in common use today, allowing technologies such as GSM to implement voice and data transport. Third generation (3G) networks are several years off; they require new and expensive equipment. 2.5G networks upgrade the speed of 2G networks while avoiding the high price tag of 3G networks. GPRS is one of the services provided by 2.5G networks.

GPRS is a data-only enhancement to GSM. Where GSM provides both voice and data transport, GPRS provides only data transport. Recall from Chapter 4 that GSM uses Time Division Multiplexing (TDM) to place digitized voice or data information into fixed size packets that get transferred in fixed time slots. Currently, this multiplexing is done over a set of circuits; each circuit represents a voice or data channel. Although the data is sent in packets, the circuits are multiplexed in order. This means that if nothing is sent over a circuit, then that slot in the multiplexed data stream is empty. GPRS uses true packet switching; all data is sent via packets exactly when they are ready. The time slots are still fixed, but each will contain a data packet if there is one available. The stream resembles a network, and network concepts such as packet reassembly and fragmentation are applicable to GPRS.

GPRS reduces overhead and increases the utilization of slots in the multiplexed stream. It takes on properties of a network—i.e., it is always on and supports concepts like virtual connections. Data over GPRS also move much faster; the upper bound on data speed over GPRS is 171.2 Kbps, versus 19.2 Kbps for GSM. This upper bound depends on the maximum number of timeslots used for GPRS packets (eight slots is the maximum); initial implementations will likely fall back to using two or three slots, thereby reducing the speed practically possible.

GPRS also transitions users from the restrictions and costs of a dial-up model of Internet access to the benefits of "always-on" network-style access. Users who use dial-up technologies like GSM are used to a delay for connection time, slower access times, and a cost model that charges for the duration of the connection. GPRS is always on and connection time is vastly reduced. The cost model for GPRS is to charge for data sent or received. This means that normal use—like reading and understanding a Web page—will potentially cost much less with GPRS.

There are downsides to using GPRS. Because it takes more timeslots than other technologies, it does not share the frequencies as well and therefore consumes more resources. Other communication methods—voice and 2G data can transmit in parallel with GPRS—can be squeezed out by too much GPRS traffic. In addition,

the use of several timeslots at once increases the probability that packets will be lost or corrupted by assembly errors. This results in greater overhead and reduced data throughput.

It is important here to reiterate that GPRS is a stopgap measure. It is intended to upgrade 2G networks by using the same equipment with new protocols. In doing so, it paves the way for 3G network implementation.

19.1.2 Using GPRS

In order to take advantage of GPRS, users need several different technologies to come together:

- *A user must have a GPRS-enabled phone and/or computer.* Existing GSM or TDMA devices do not support GPRS.
- *A user must subscribe to a carrier service that implements GPRS.* GPRS-enabled service is just now (in 2002) being implemented and is in its infancy.
- *GPRS must be enabled for a particular service.* Because data and voice services can be run in parallel with GPRS, it is possible that a user will have voice service and not data service. Users need to be specifically enabled for GPRS data use.
- *There must be a GPRS-enabled destination.* In addition to mobile devices, Internet destinations are reachable through GPRS (we call these "GPRS-enabled").

It is important to think of GPRS as analogous to TCP/IP or WAP and not equivalent to something like GSM. Web pages are accessible over GPRS, for example, as are other data services, like email and printing protocols.

19.1.3 Symbian OS Support for GPRS

The support that Symbian OS needs to give to GPRS is minimal. Currently, version 6.1 treats GPRS in the same way as it handles GSM: both are considered to be call-oriented connection media. In version 6.1, devices can receive SMS over GPRS, and GPRS is integrated into ETel. However, it is not yet viewed like a network protocol; there are no PRT modules implemented yet. This means programmers cannot use GPRS for data link layer transport and users with GPRS data service cannot use it transparently as they would services like network transport. These will undoubtedly be coming in future versions.

19.2 Internet Protocol version 6

Currently, there are only 4,294,967,296 possible IP addresses. While this may seem like a lot, when you take out the reserved addresses (e.g, for a local host or multicasting) and the addresses used for broadcasting, the actual number that can be assigned to devices is substantially reduced. If you consider how many devices might possibly be assigned an IP address (e.g., it is estimated that there were 860 million mobile phones worldwide at the end of 2001, and that this number will double in five years), this number dries up quickly. Even with schemes like dynamic address assignment, IP numbers are starting to become scarce.

Most of today's Internet uses version 4 of the Internet Protocol (IPv4). IPv4 is now nearly twenty years old and has been remarkably resilient in spite of its age. But recently, it has developed some problems. First and foremost, there is the growing shortage of IPv4 addresses. Second, as more devices proliferate across many different carriers, routing algorithms have grown in complexity and have tested the bounds of IPv4 routing capability. Third, more protocols on more devices have pushed the limits of fixed headers in IP packets. Finally, issues of autoconfiguration and self-renumbering have cropped up as beneficial to system administrators and users alike.

Internet Protocol version 6 (IPv6) has been designed to address these situations and more. The most noticeable change with IPv6 is the size of its addresses. IPv6 addresses are 128 bits wide, versus 32 bits for IPv4. The notation is a colon-delimited notation, such as `A:B:C:D:E:F:G:H`, where each letter is a 16-bit quantity. This address space now contains 2^{128} addresses, or 3.4×10^{38}. Because the address space is so large, we can create additional addressing hierarchies. IPv4 addresses are split into network ID and host ID portions. IPv6 could add a new level of Internet Service Provider to these existing levels. IPv6 routing might first route a packet according to the ISP ID, then route to a network within the ISP, and finally route to a host.

IPv6 also specifies a flexible header format to packets. Many headers are optional and can be omitted. This accommodates both lower-bandwidth networks and networks with large capacity that want more features built into each packet. IPv6 is open-ended with its protocol definition. More protocols can be added in the future and can be incorporated into the protocol structure; underlying network hardware could change without harming the protocol stack. IPv6 also includes support for the autoconfiguration and dynamic renumbering of a machine's address. A new machine could join an IPv6 network by assigning itself an address and changing that

address on the fly as it participates in network traffic. IPv6 has also been specified to incorporate other protocols' addressing schemes.

To use IPv6 on a network, all devices on that network must be using IPv6. Unfortunately, IPv4 is incompatible with IPv6. IPv6 can incorporate some IPv4 features—e.g., an IPv4 address is also an IPv6 address—but many facets of IPv4 do not translate to IPv6. So, new network equipment, new drivers and new software are all required to take advantage of the new IP mechanism.

Fortunately, Symbian OS version 6.1 is IPv6 aware. Many of the APIs that have network interfaces can work with both IPv4 and IPv6. The protocol modules still need to be upgraded; however, compatibility can be easily reached with the inclusion of a future `tcpip.prt` implementation.

19.3 Other Future Technologies

It is a very safe prediction that GPRS and IPv6 will be incorporated in future versions of Symbian OS. There are other communication technologies being developed whose future is uncertain and therefore whose support from Symbian OS is not a given.

19.3.1 MMS: Multimedia Messaging Service

A message—as we have discussed it in this book—is usually a text-centered object with data attachments. These data attachments can be any type of object and are accessed by using a viewer for the attachment. The model for this type of messaging is one that centers on text messages with connected multimedia pieces that must be detached and viewed.

A new standard for messages takes the opposite viewpoint. The Multimedia Messaging Service (MMS) lets users send and receive messages with formatted text, graphics, images and audio and video clips. The model for an MMS message is a data stream rather than a set of text. It supports image formats (e.g., GIF and JPEG formats), video formats (e.g., MPEG 4) and audio formats (like MP3).

In the MMS model, messages can come from many different sources. They can be downloaded to a device from a WAP site, transferred via an attached device (such as a digital camera) or received as an MMS message. MMS messages can be sent either to another MMS-enabled mobile phone or converted to email and sent to an e-mail address.

The MMS model does not specify transport but, for many formats, it is dependent on networks with high transmission speeds. GPRS

and 3G technologies (see Section 19.3.3) are necessary for most uses. GSM can be adapted to support MMS, but the slow data rate of the technology will restrict the formats that can be sent or received. WAP is used for transport over GSM.

Several companies have begun to implement MMS compatible devices. The Nokia 7650 is MMS-capable and the Series 60 development platform for Symbian OS has MMS support built in. Ericsson's T68 phone has MMS support and the company also has adapters that will make non-MMS phones able to receive MMS messages.

19.3.2 i-Mode Service

i-mode is a communication suite dedicated to allowing mobile phone access to Internet protocols. It is wildly popular in Japan but currently only in Japan. In October, 2001, there were over 27 million i-mode subscribers, and this number was increasing by about 40,000 per day. There were about 60 million mobile users in Japan, which means that i-mode accounted for about 47% of mobile phone usage. At the moment 99.999% of i-mode users are Japanese. Since i-mode has only been in use since early 1999, these numbers are quite impressive.

i-mode was invented and is currently supported by NTT DoCoMo, a subsidiary of NTT, Japan's national telephone carrier. At the present time, NTT DoCoMo is the sole carrier of i-mode data.

Just as GPRS is a packet-switched extension of a GSM or TDMA phone system, i-mode is the packet-switched extension to NTT DoCoMo's ordinary mobile phone system. As with GPRS, i-mode is "always on". This means that it is (in part) a data link layer carrier with unique properties for use on mobile phones. For example, packet sizes are fixed at 128 bytes. Overhead reduction is important, because the current speed of i-mode is only 9600 bps.

i-mode also addresses content. i-mode uses cHTML (compact HTML), a subset of HTML that represents a new HTML standard. In addition to HTML tags, i-mode defines additional tags that are used only in i-mode microbrowsers. For example, there are new HTML tags that can define a link that dials a phone number when it is clicked and ones that identify Web pages as i-mode pages. There are also many special characters used in i-mode phones, such as symbols for kisses and hot spring baths.

A comparison between i-mode and conventional phone services like WAP is not simple. At the present time, WAP is implemented over a call-based carrier while i-mode is implemented over a data link network carrier. WAP uses WML for its contents, while i-mode

uses cHTML (which is perhaps easier to learn). However, when a comparison is made between WAP using GPRS and i-mode, the two are much closer in function. Currently, GPRS has the transfer speed advantage over i-mode.

Whatever promise i-mode shows in its popularity and usefulness is hampered by its proprietary nature. At the present, NTT DoCoMo is the sole implementor of i-mode, and there have been no attempts to make i-mode a standard. Until this changes, it is unlikely that we will see i-mode in wide use outside of Japan.

19.3.3 3G Mobile Phone Service

One of the constant pressures on mobile communication providers is for faster transmission speeds. Digital transmission over current 2G networks is bounded by 19.2 Kbps; GPRS running over 2.5G networks has a theoretical upper limit of 171.2 Kbps. While 2.5G is an improvement, it does not approach the speeds necessary for many of the applications planned for mobile communications. Video streaming, for example, needs network-level bandwidth.

Third generation (3G) mobile radio communications will help meet these bandwidth needs. 3G networks will provide data in speeds up to 2 Mbps. For 3G communication, GSM and GPRS protocols of the past will be replaced with Universal Mobile Telecommunication System (UTMS) protocols. UTMS is a radio communication standard that provides two different transmission implementations:

- *Frequency Division Duplex (FDD)* is a method that uses two radio frequencies to transmit data: a *downlink* from the provider to the client and an *uplink* from the client to the provider. This method is best used in larger broadcast cells for general purpose use. The maximum data speed in FDD environments is 384 Kbps.
- *Time Division Duplex (TDD)* uses a single frequency, transmitting down and uplinks in fixed time slots. Less effort is spent maintaining a pair of frequencies, but there are higher error and transmission rates. Therefore, TDD must be used in much smaller cells, where the receiver is closer to the transmitter and is not moving. In these environments, data rates approach 2 Mbps.

TDD has a further advantage in that it can be asymmetric in transmission. The amount of data in the uplink can be very different to the downlink; the 2 Mbps bandwidth would be filled by whatever side needed to transmit the most. This means that TDD is better suited

for contexts where a simple request results in a large download of data (e.g., requesting a video or audio stream).

While it is possible to run GPRS over existing GSM equipment, UTMS requires entirely new installations. To implement 3G networks, providers are going to have to roll out brand new services to brand new customer equipment. New phone handsets that support UTMS will be needed. However, once the equipment is in place, new, all-digital services can be used. Voice-over-IP (VOIP) services that encode analog voice transmission for sending out over IP networks can be enabled. Services that implement IP networking over UTMS carriers will support efforts like VOIP. Messaging services over high speed connections now take on new meaning when the carrier services approach network speeds.

3G networks will increase call capacity, provide network services to handheld devices, and increase transmission speed. However, to Symbian OS, it will be the carrier that lies underneath much of the communication architecture. The biggest change for Symbian OS designers to consider will be how 3G will affect services and applications which are already supported. If we can send data at 384 Kbps in an always-on network context, how does that affect SMS messaging? Do we really want short messages in this context? How does this affect SMS-based data, like WAP content? Providing services in an environment where the quality of networks change will also be challenging. More expensive 3G networks will probably only be available initially in dense population centers with 2G and 2.5G network coverage elsewhere. Providing consistent application service through service changes and interruptions will be extremely difficult.

19.4 Summary

The word "uncertain" is a good way to summarize future technologies. This chapter muses about a few technologies of the future and how Symbian OS might support them. However, the future is prone to change. While GPRS and IPv6 are almost certain to be used in the future, other technologies change as they are implemented. Only time will tell.

Appendix

Developer Resources and Bibliography

Download the code for the examples in this book from ***http://www.symbian.com/books/socp/socp-support.html***.

Documentation for the various releases of Symbian OS can be found in the SDKs for each release.

White papers on all aspects of Symbian OS can be found at ***http://www.symbian.com/developer/techlib***.

Symbian Web Sites

Corporate: ***http://www.symbian.com***
Developer resources: ***http://www.symbian.com/developer***

Symbian Partner Developer Web Sites

Ericsson: ***http://www.ericsson.com/developers***
Nokia: ***http://www.forum.nokia.com***
NTT DoCoMo: ***http://www.nttdocomo.com***

Chapter-by-chapter resource guide

Chapter 1 An Introduction

Allin, J., *et al.*, *Wireless Java for Symbian Devices*, John Wiley and Sons, Ltd., 2001.

The ENIAC: ***http://ftp.arl.mil/mike/comphist/eniac-story.html***

Nokia phones: ***http://www.forum.nokia.com***

Chapter 2 An Introduction to the Symbian Communication Architecture

Pottruck, D.S. and Pearce, T., *Clicks and Mortar*, Jossey-Bass Publishers, 2000.

Chapter 4 Transport Technologies

Comer, D.E, *Internetworking with TCP/IP: Principles, Protocols, and Architectures*, Volume 1, Fourth Ed., Prentice Hall, 2000.

Tutorials on RS232: ***http://www.arcelect.com/rs232.htm***
http://www.sangoma.com/signal.htm

Network cabling: ***http://www.techfest.com/networking/lan/ethernet5.htm***

IrDA standards: ***http://www.irda.org***

Bluetooth protocol standards: ***http://www.bluetooth.org***

Chapter 5 A Look at Content Technologies

Email standards (including receipt and delivery): ***http://www.ietf.org***

SMS messaging: ***http://www.iec.org/online/tutorials/wire'sms/***

vCard and vCalendar formats: ***http://www.imc.org***

Fax protocol standards: ***http://www.faqs.org/rfcs/rfc804.html***

HTTP and related standards: ***http://www.ietf.org***

HTML standards: ***http://www.w3c.org***

WAP protocol standards: ***http://www.wapforum.org***

Chapter 6 Security and Communication

Denning, D.E., *Information Warfare and Security*, Addison Wesley, 1999.

Kaufman, C., Perlman, R., and Speciner, M., *Network Security: Private Communication in a Public World*, Prentice Hall, 1995.

Security attack information: ***http://www.cert.org***

Chapter 8 Serial Communications

Tasker, M., *et al.*, *Professional Symbian Programming*, Wrox Press, 2000.

Chapter 10 Using Sockets

Stevens, W.R., *UNIX Network Programming*, Prentice Hall, 1990.

General socket programming:
http://www.uwo.ca/its/doc/courses/notes/socket/
http://compnetworking.about.com/library/weekly/aa083100a.htm
http://world.std.com/ jimf/papers/sockets/sockets.html
http://py-howto.sourceforge.net/sockets/sockets.html

Chapter 12 Communicating with TCP/IP

IP address classes and routing: ***http://www.ralphb.net/IPSubnet/***
Network Time Protocol: ***http://www.eecis.udel.edu/ntp/***

Chapter 17 Synchronization: PLP and SyncML

SyncML: ***http://www.syncml.org***
Developing with SyncML: ***http://www.nokia/com/syncml***
http://www.forum.nokia.com/main/1,6668,1˙47,00.html

Chapter 18 The Communications Database

Bowman, J., Emerson, S., and Darnovsky, M., *The Practical SQL Handbook*, Addison Wesley, 2001.
Kaufman, J., Matsik, B., and Spencer, K., *Beginning SQL Programming*, Wrox Press, 2000.

Chapter 19 Looking Ahead

GPRS: ***http://www.gsmworld.com/technology/gprs.html***
IPv6: ***http://www.ipv6.org***
MMS: ***http://www.ericsson.com/mms/***
http://forum.nokia.com/main/1,6668,1˙2˙7,00.html
i-mode: ***http://www.eurotechnology.com/imode/faq.html***
3G networks: ***http://www.3gnewsroom.com***

Index

Active object 29
  Use 166
Addressing definitions 196
AgendaFileReader class 175
AGENTNotifier class 244
agt file 215
Application CSYs 37
AppUi class 145
Asynchronous events 29
Attribute lists 251
Authentication
  Drawback 114
  Uses 115
Authentication process
  Identification 114
  Private information 114

Baseband 73
Binding server sockets 258
bldmake, 130, 137
Bluetooth connection, opening 266, 271
Bluetooth device services 236
Bluetooth instant messaging
  Designing the application 262
  Initializing the application 264
  Sender object 265
Bluetooth model 235
Bluetooth radio 72
Bluetooth socket, opening and configuring 256
Bluetooth, initializing the system 255
Bluetooth, system definitions 238
BTIMReceiver class 270
BTIMSender class 265
Building the application
  for a target machine 132
  the command line, using 132
  visual c++ 131

CActive class 162
CAgnModel class 175
Call, answering 292
Call, opening 289
Call, ownership 294
Call, terminating 295
CBaseMtm class 302, 315, 316, 321
CBaseMtmUi class 302, 312
CBaseMtmUiData class 302
CBaseServerMtm class 303
CClientMtmRegistry class 307, 308
CCommsDatabase class 363, 364
CCommsDatabaseBase class 363, 372, 376
  BeginTransaction() 372
  CommitTransaction() 372
CCommsDbTableView class 365, 369
Cellular networks 276
Characterizing service attributes 249
Class constructor 174
Class settings 254
Client-Server relationships 28
  Clients 40
  Kernel servers 37
CMsvSession class 306, 308
  OpenAsyncL() 306, 308
  OpenSyncL() 306, 308

Communication phases
  Closing the communication device 142
  Configuring the communication device 142
  Exchanging data 142
  Initialization 141
  Opening the communication device 142
Communications database classes 363
Communications database server 39
Configuration and device discovery 266
Configuration files, building 130
Configuration phase 349
Connected sockets, configuring 184
Connecting client sockets 257
Connection manager 8
Connectionless sockets, configuring 183
Connectivity servers 8
Constructors and deconstructors 224
Content Technologies
  WML and WAP 25
  HTML 25
  Messaging 24
  World Wide Web 25
CPullerAppUi
  DisconnectFromNetwork() 223
  IsNetworkOpen() 222
CSdpAgent class 242
  AttributeRequestL() 242
  NextRecordRequestL() 242
  SetAttributePredictorListL() 242
  SetRecordFilterL() 242
CSdpAttrIdMatchList class 245
CSdpAttrValue class 249
CSdpAttrValueUUID class 250
CSdpSearchPattern class 243, 244
CTimer class 159

Database Management System (DBMS) 362
Database object, creating 364
Database structure 363
Database, creating a view 365
Debugging hardware signals 55
Denial of Service attack (DoS) 117
Device and service, setting explicitly 240
Discovering a device 240
Discovering services 241
Distributed Denial of Service attacks (DDoS) 117
DNSServerSocket.Open 218
Domain Name Service (DNS) 191

Electronic mail 86
Emulators, hardware connections
  Function keys 134
  GSM access 135
  IR port 135
Encryption techniques 111
Entry, properties
  Entry ID 309
  file storage 309
  Header information 309
  MTM list 309
  Owning service 309
ETel model, components
  Call object 289
  Line component 286
  Phone component 282
  Root server 279
Ethernet 64
Exchanging data 155, 204, 258
Exchanging data over sockets 187
  Closing sockets 188
  Connected sockets 188
  Connectionless sockets 187
Extended Markup Language (XML) 356

File converters 8
File data, transferring 98
Finishing configuration
  Binding server sockets 219
  Connecting client sockets 218

Frequency Division Duplex (FDD) 385
FTP 105

Gateway server 28
Generic API 31
GPRS, Symbian OS support 381
GPRS, using 381
GPRS, what is 379

Handshaking 55
Host resolver 220
HTML, use 101
HTTP:hypertext transfer protocol 98

i-Mode service 384
IAS class 205, 206
ICMP packet 117
ICMPSocket.Open 218
IMAP4
  Internet Message Access Protocol 90
Infrastructure issues
  Security concerns 44
  Synchronization 43
Infrastructure modularity 37
Initializing the system 149
Internet Access Points (IAPs) 374
IrCOMM 61
IrDA 55
  link access layer (IrLAP) 57
  Link management layer (IrLMP) 59
  physical layer 56
  Programmer API 60
  Tiny transport protocol (TinyTP) 60
  socket options 197
IrLAN 60, 61, 67, 68
IrLMP implementation, services
  Connection 59
  Data delivery 59
  Device discovery 59
  Disconnection 59
IrOBEX 61
IrTranP 61
iServiceSocket.Open 220

Java 2 Micro Edition(J2ME), framework 8
Java Virtual Machine implementation 8

KAfInet address 214

L2CAP 74
L2CAP layer, services
  Data stream segmentation and reassembly 74
  Group abstractions 74
  Protocol multiplexing 74
Landline networks 276
LMP
  link manager protocol 73

makesis 133
Making a call 291
makmake 125, 131
Message characteristics
  Content 84
  Destination 84
  Format 84
  Sender 83
  Timestamps 84
Message event 327
Message server functionality 304
Message server, functions
  Access to MTMs 304
  Access to the message data repository 304
Message, BIO 24, 94, 331
Message, dissecting 300
Message, Email 24, 329
Message, Fax 25, 95, 331
Message, receiving 272
Message, sending 268
Message, SMS 24, 92, 330
Messages, accessing 308
  Creating a session 306

Messages, accessing (*continued*)
  Opening the registry 307
  Other operations 312
  Viewing and editing messages 310
MMP project file 128
Multimedia messaging service (MMS) 383
Mobile phones
  Keyboard-based 11
  Pen-based 12
  Single-handed use 11
  Smartphones 11
Models for mail delivery
  Pull model 85
  Push model 85
Module
  CSY 172
  LDD 172
  PDD 172
MSdpAgentNotifier class 242, 243, 245
  NextRecordRequestComplete() 242
MTMs 302
MTM, classes
  Client-side MTM 302
  Server-side MTM 303
  User interface data MTM 302
  User interface MTM 302
MTM-specific functions 314

NetDial agent 215
NetSpy 109
Network, connecting 212
Notification of changes 375
Notifications and call information 293

Other stack components 76

Page grab, starting 227
Page grabber, cancelling 225
PageGrabber active object 226
Pagegrabber state diagram 225
PageGrabber class 223
  ConstructL() 225
  DoCancel() 225
  IssueRequest() 227
  NewL() 225
  NewLC() 225
  RunL() 228
Parts list 312
Password, choosing the correct one 115
Phone lines, initializing 288
PhoneCall 285
POP3
  Post Office Protocol 89
PPP, main features
  Framing method 67
  Link control protocol 68
  Option negotiation 67
Pretty Good Privacy (PGP) 119
Promiscuous mode 108
Protocol implementation modules
  CSY modules 34
  MTM modules 35
  PRT modules 35
  TSY modules 35
Protocol, initiating 202
Protocol, shutting down 204
Psion Data Port 51
Psion Link Protocol (PLP) 62, 353

RBTMan class 259, 261
RComm 160
RComm class 152, 153, 155, 160, 161
RCommServ 160
RCommServ class 148
Receiver connection, initializing 270
Receiver object 270
Receiver socket, configuring 271
Records, accessing and reading 369
Records, creating and editing 370
Records, deleting 375
Records, hidden 375
Remote control, using Symbian OS device 57

RFCOMM 75
RHandleBase class 159
RHostResolver class 190, 191, 199
RLine class 287
RNetDatabase class 205, 206
RNotifier class 267
RollbackTransaction() 373
Root server
  Connect() 279
RPhone class functions 282
RSdpDatabase class 248, 252
RSessionBase class 247, 261
RunL(), implementing 228
RWAPConn class 338
RWAPServ class 337
RWDPConn class 339–341
RWriteStream class 175

SDP 75
Secure HTTP (SHTTP) 119
Security attacks 116
Security issues
  Data issues 110
  Physical issues 108
  Remote access control 112
  Threats to Symbian OS devices 113
Serial communications server 39
Serial port 135
  Closing 159
  Configuring 153
  Opening 152
Servers, access by clients 38
Servers, services layer
  Generic Custom Service (GENCSERV) 353
  Link server 353
  Remote File Services (RFSV) 353
  Remote Protocol Command Service (RPCS) 353
Service attributes, searching 245
Service discovery agent 241
Service IDs, searching 243
Signal
  Clear To Send (CTS) 54
  Data Carrier Detected (DCD), 54
  Data Terminal Ready (DTR) 54
  Receive Data (RD) 54
  Request To Send (RTS) 54
  Transmit Data (TD) 54
Simple attribute types 249
SIrdaAddr structure 196, 200
  iFirstServiceHintByte 196
  iSecondServiceHintByte 196
  iSniff 196
Simple Mail Transfer Protocol (SMTP) 88
Sniffing 196
Socket Properties 181
Socket configuration, choosing 217
Socket server 39
Socket usage, patterns 182
Socket, closing 259
Socket, getting information 221
Sockets, configuring and connecting 183
Software Development Kits (SDKs)
  EPOC release 5 123
  EPOC release 5 connectivity 123
  Symbian OS v6.1 Connectivity 124
SOURCEPATH 129
Special purpose tables 374
Specification files
  Help system files 127
  Package construction files 127
  Project files 125
  Resource files 125
State diagrams 163
Storage and message structure 305
Structured Query Language (SQL) 365
Symbian OS 4, 6, 8, 10, 12, 14
  Application engines 6
  Framework, user interface 7
  Kernel 5
  Middleware 6
  Synchronization technology 8
Symbian OS communications infrastructure
  Device driver layer 34

Symbian OS communications infrastructure (*continued*)
Physical device 34
Protocol implementation layer 34
Synchronization mechanics
Communication media 351
Data objects, files, and timestamps 350
Protocol requirements 351
Synchronization process
Synchronization phase 350
SyncML 8, 63
characteristics 356
SyncML tags
CmdID 358
Cred 358
Item 358
Meta 358
SyncML, protocol stack 358
SyncML, support 359

TARGETPATH 129, 137
TBTDevAddr class 238
TBTServiceSecurity class 259
TCallInfo structure 287
TCommConfigV01 class 153
TCP/IP 65
protocol review 212
socket options 221
system definitions 213
Telephony device, queries
Device capabilities 283
Information about calls opened on a line 286
Information about the phone lines 284
Line capabilities 286
Name of the call, status, capabilities 287
Status of a line 286
Status of the call going across a line 284
Status of the telephony device 283
Telephony server 39
Telephony server functions
EnumeratePhones() 279
GetPhoneInfo() 279
TVersion Version() 279
TELNET 105
The Service Discovery Database 247
3G mobile phone service 385
TIASQuery class 205, 206
Time Division Duplex (TDD) 385
Timeout, asynchronous programming 29
TInetAddr object 218–221, 229
TInquirySockAddr class 254
TIrdaSockAddr class 196, 200
TLineInfo structure 284
TNameEntry class 193
TNameRecord class 192
TNifProgress class 216
TodoX
Application, building and running 136
Exchanging to-do list items 137
Source files, editing 139
TodoXferBase class 144
TodoXferIRSerial class 173
TodoXferIRSocket class 202, 203
Close() 205
Configure() 199
OpenL() 199
ReceiveItem() 203
Send() 204, 209
SendItem() 202
TodoXferIRSocket class
ConfigureReceiver() 201
TodoXferSerial class 150–152, 154, 159, 166
TodoXferSerialAO class 166
TPhoneInfo structure 279
TProgress structure 296
TProtocolDesc class 191
Transactions 372
Translating state diagrams to active object code 163
Transport technologies
Bridge technologies 23

Transport technologies (*continued*)
  Networking technologies 21
  Point-to-Point transport methods 19
TSerialInfo class 160
TSockAddr class 184, 257
TSY modules
  LoadPhoneModule() 279
  UnloadPhoneModule() 279
TUUID class 243
Types of attacks relevant to small devices
  Attacks on authentication 116
  Attacks on protocols 117
  Attacks on services 116

Unique UID Numbers, obtaining 130
UnitInvoke() function 341
UnitWaitResults() function 341
Universal Mobile Telecommunication System (UTMS) 385
User class 149, 156
Using the emulators
  Bluetooth 135
  communication devices, with 135

vCards 95
View records, navigation
  GotoFirstRecord() 369
  GotoNextRecord() 369
  GotoPreviousRecord() 369

WAP
  Bearers 335
  Session server 336
  Stack support 336
WAP content, viewing 345
Watchers 299
wCClientMtmRegistry class 307
WDP layer, implementing 339
Web technologies, Symbian OS support 25
WebServerSocket.Open 218
Window server 6
Wired media 50
Wireless media 52
WML, use 103
WSP layer, implementing 340
WSP sessions, connected 342
WSP sessions, connectionless 340
WSP: wireless session protocol 102

Symbian OS Communications Programming

Michael J Jipping

Communication lies at the heart of Symbian OS and the mobile phones that use it. Symbian OS has a powerful and flexible communications architecture that addresses current state-of-the-art as well as future functionality. This rich environment gives developers access to important technologies such as wireless telephony, Bluetooth, TCP/IP networking, infrared data transport, email and SMS messaging, and WAP content.

Please visit Symbian Press at:

www.symbian.com/books

www.wiley.com

ISBN 0-470-84430-2

Symbian OS Communications Programming is a guide to the Symbian OS communications architecture. It provides developers with authoritative and practical information on the communications models and programming interfaces that are used by Symbian OS, along with examples.

SOCP covers all transport technologies supported by Symbian OS v6.1 including: serial and infrared communication; TCP/IP network support; Bluetooth; and telephony. Examples show how to take advantage of each technology.

SOCP also provides coverage of content technologies, such as HTML and WAP, and includes an explanation of the Symbian OS messaging architecture. Examples demonstrate the sending and receiving of email and SMS messages and show the client/server interactions involved in pushing and pulling content to and from servers.

Advanced topic coverage includes the use of the Symbian OS communications database and a review of supported and future synchronization technologies such as SyncML.

This guide is designed for developers who have experience of programming in C++ and are interested in taking advantage of the communications possibilities of Symbian OS. No prior experience with communications programming is assumed.

Dr. Michael J Jipping is Associate Professor of Computer Science at Hope College, Holland, Michigan, USA. He is a long-time member of the Symbian Developer Network, and has been developing software for Symbian OS for over seven years. He is actively involved in research on handheld networking and has co-authored several textbooks on programming languages.

Source code for all examples are available for download from the accompanying website at www.symbian.com/books

WILEY
wiley.com