

# 2

## Overview

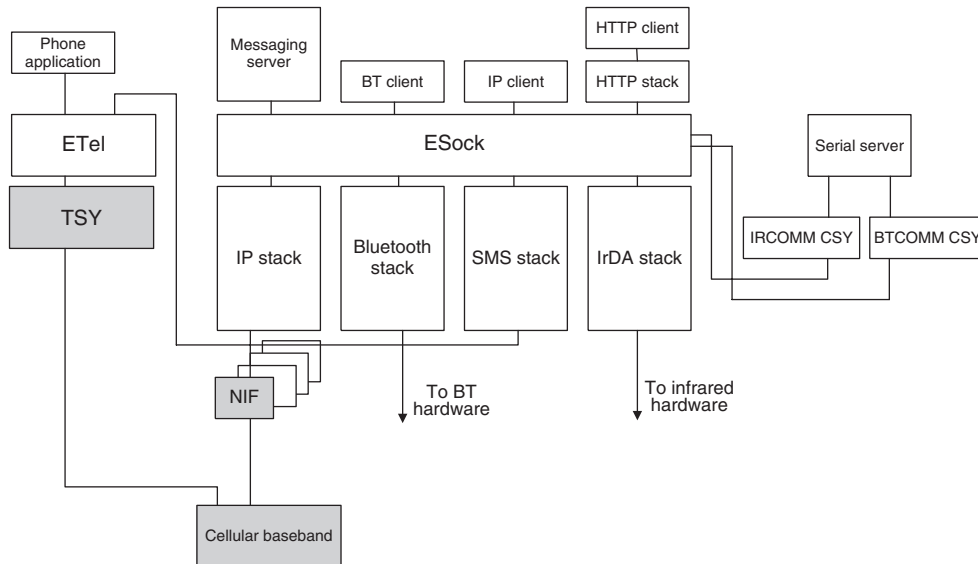
In this chapter we'll present a brief overview of the different parts of Symbian OS we're going to talk about in this book. Detailed explanations of how things work will be left to the individual technology chapters, but here we'll take a quick look at what each area does.

First we have something of an aside, but on an important topic – that of binary compatibility (BC).

Symbian OS v9.0 introduced a major BC break with versions 8.1 and earlier, due to the introduction of a new (standardized) application binary interface (ABI). Therefore applications compiled against v8.1 and earlier of Symbian OS will not run on v9.1 and later. In addition to this, the platform security model requires some changes to applications using communications functionality. Depending on the application this might be as simple as adding a CAPABILITY statement to the MMP file.

To start with we take a quick look at a very high-level model of how a Symbian OS-based phone is constructed in order to see where various components fit in. This isn't to say every phone looks like this, but it is a good enough model to help understand the different layers that exist within Symbian OS.

The signalling stack (also referred to as the 'baseband' or 'cellular modem') is provided by the manufacturer of the device, so Symbian OS contains a series of adaptation layers to allow standard Symbian OS APIs to be adapted to the specific signalling stack in use. Today, all shipping products use either a 2.5G GSM/GPRS or 3G UMTS (aka W-CDMA, aka 3GSM) signalling stack. In Figure 2.1, you can see the layers that form the adaptation coloured grey – we'll talk a little about each of these throughout this chapter.



**Figure 2.1** High-level overview of Symbian OS communications architecture

## 2.1 Low-level Functionality

We'll split our overview into two main sections, following those that the whole book uses, in order to partition the system more clearly. In this first part of the chapter we'll talk about the technologies in section 2 – the lower level areas such as the Bluetooth, IrDA and IP stacks, along with telephony. In the second part we'll discuss more about messaging, OBEX, HTTP and OMA Device Management.

There are a couple of key frameworks involved in Symbian OS communications, so let's look at those first. Not everything fits into these frameworks (most of the higher level functionality in section 3 does not, for example) but they provide the foundation for the topics discussed in section 2.

Symbian OS undergoes continual development, so details on the internals are subject to change. Public interfaces (published in Symbian parlance), which make up most of the content of this book, are subject to strict compatibility controls; however, internals are likely to evolve over time. When we're discussing internal details, which we are going to do in this chapter, we'll make it clear so you know that the information may change.

## 2.1.1 The Root Server and Comms Provider Modules

Root server and comms provider modules are internal to Symbian OS<sup>1</sup> – therefore this whole section should be considered informational.

Many of the lower level communication technologies in Symbian OS run inside the c32exe (often abbreviated to 'c32') process. In the initial versions of Symbian OS (or EPOC32, as it was back then) the serial server<sup>2</sup> was the initial thread in this process, and provided functionality to start further threads within the c32 process. Today, the root server performs this function, loading and binding comms provider modules (CPMs) into the c32 process. The serial server, ESOCK server and ETel server are all now loaded as CPMs by the root server. All this is done under the control of the c32start process, which is responsible for loading and configuring the CPMs based on the contents of various CMI files found within c32start's private directory.

The exact mechanisms used to load ESOCK, ETel and the serial server aren't normally of any interest to us. However, as we'll see in Chapter 13, there are some circumstances when using the emulator when we might want to alter which CPMs get loaded, so it's useful to keep the above information in mind when setting up development environments for communications programming.

## 2.1.2 ESOCK

ESOCK provides both the framework for, and the interface to, various lower level protocols in Symbian OS. For example, core parts of the Bluetooth stack, most of IrDA, most of the TCP/IP stack, and much of the SMS stack<sup>3</sup> exist within the ESOCK framework, and are accessed through ESOCK APIs. Figure 2.2 gives an overview of the ESOCK framework. Chapter 3 contains a lot of useful information about the public ESOCK APIs, with additional information in the technical chapters that refer to technologies implemented as ESOCK plug-ins, such as Bluetooth. As you can guess from the name, one of the key public interfaces that ESOCK provides is based upon a sockets API.

Here we'll talk a little about what happens behind the scenes with ESOCK and the various plug-ins.

---

<sup>1</sup> Actually, the APIs are available to Symbian partners, but for the purposes of this book, that's the same as internal to Symbian.

<sup>2</sup> 'Serial server' is also quite a recent term – originally it was called the c32 server – a reflection of the fact that the hardware available for communications in the original Psion Series 5 consisted of a serial port and an IrDA port, and therefore the serial server was the interface to the core communications functionality in the device.

<sup>3</sup> Although there are a variety of places in the system where you might choose to access SMS messages – see Chapter 8 for details.

Again, this is information about internals, so is subject to change.

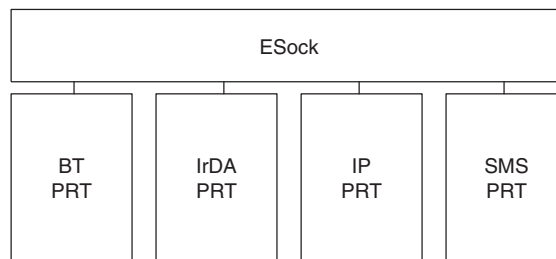
ESOCK plug-ins exist in DLLs with the suffix .prt, and are loaded into the ESOCK threads as part of the ESOCK startup mechanism. Specifically, ESOCK looks for ESK files in the ESOCK subfolder of the c32 private directory. These contain details of how to bind various protocol layers together, along with some protocol-specific configuration information. As with CMI files, we'll see some more details of this in Chapter 13 when it comes to setting up development environments.

## **Bluetooth**

The Bluetooth standards include a rich and varied set of functionality, ranging from using wireless audio headsets for making telephone calls, through emulating an Ethernet network, to connecting keyboards to a host. Much of the functionality available is described in Bluetooth profiles – documents describing an end-to-end use case for a user, such as using an audio headset with a mobile phone as a hands-free kit. Those use cases tend to be implemented completely by the time a phone reaches the market. However, there are also APIs available to application programmers to implement their own Bluetooth-based applications and protocols.

Chapter 4 describes two main areas – what is required to extend a profile that Symbian implements (for example, interfacing to the remote control functionality of the Audio/Visual Remote Control Profile (AVRCP) profile to allow your application to receive commands from a Bluetooth stereo headset) and what is required to implement Bluetooth communications between devices running your application, using one of two possible protocols – L2CAP or RFCOMM.

There are also some Bluetooth profiles which can be used without any Bluetooth-specific APIs. The prime example of this is Bluetooth PAN profile, which implements a virtual Ethernet network between Bluetooth devices. In this case, the standard APIs used for creating IP connections are used, although there are some special behaviors implemented to cope



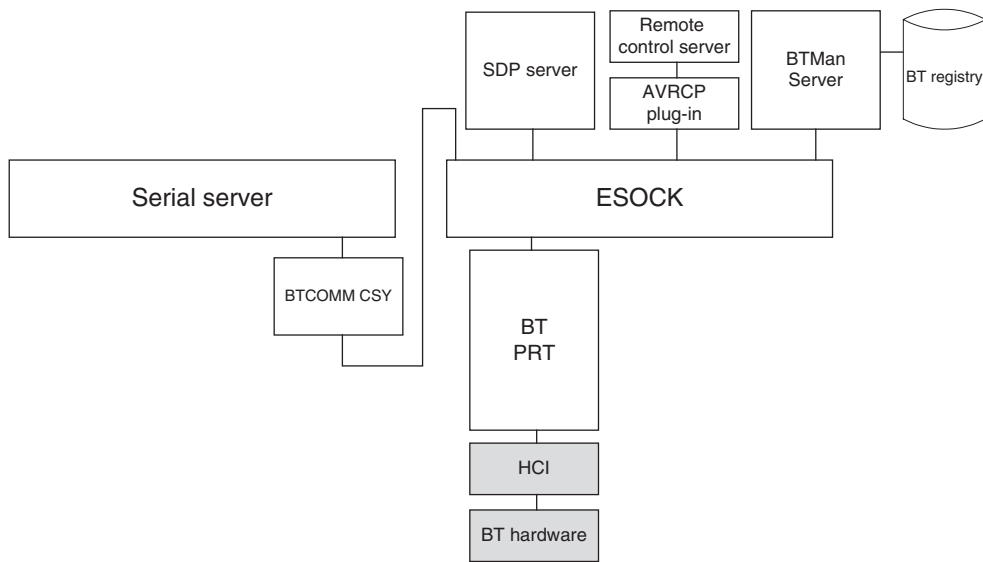
**Figure 2.2** ESOCK and protocol plug-ins

with the different models used by Bluetooth and normal, wired, Ethernet networks.

Bluetooth functionality is implemented in several modules, which are split across various processes, as shown in Figure 2.3 – here we cover the most obvious ones from a developer viewpoint. The core Bluetooth stack, implemented as an ESOCK plug-in, provides the L2CAP, RFCOMM, AVDTP and AVCTP protocols. The local service discovery database is implemented as a standalone component, which sits above the Bluetooth stack. The Bluetooth manager is a Symbian OS server that provides access to the local device's Bluetooth registry. And finally the RFCOMM CSY is the plug-in to the serial server that allows access to RFCOMM through the RComm interface.

One related component is the remote control server, RemCon, which implements a generic system for distributing commands from remote devices, typically some form of remote control, to local applications. It's mentioned here as it is the server for the AVRCP implementation, which receives commands from the remote control buttons present on some Bluetooth stereo headsets. These are then routed through a UI platform-specific target selector plug-in (TSP) to the appropriate application for processing. There are more details on this system in Chapter 4.

There are some components that we haven't mentioned, most notably the host controller interface (HCI) which provides the interface from the Bluetooth stack to the hardware, but these are adaptation specific and typically do not present APIs for general use.



**Figure 2.3** High-level bluetooth architecture

## ***IrDA***

IrDA is an older technology than Bluetooth, providing line-of-sight communications over a range of around one metre. The specifications themselves define fewer profile details than Bluetooth – the major profile being OBEX, which is covered in a separate chapter. As such, the Symbian OS support is primarily centred on the core specifications. When using IrDA, the two main protocols of interest are IrCOMM, for serial port emulation, and TinyTP, for stream-style communication. IrDA also provides device discovery and simple service discovery capabilities. Chapter 5 describes the implementation of the IrDA protocols in more detail.

While the IrDA committee has defined extensions to the core IrDA specifications, Symbian OS only provides the basic services. Key omissions are support for Fast Infrared, Fast Connect and IrSimple.

## ***IP and network interfaces***

Some of the details of the internals of the networking subsystem, for example network interfaces for particular network technologies, and the IP hooking mechanism, are internal (actually partner-only again) and therefore subject to change.

Symbian OS contains a hybrid IPv4/IPv6 stack, implemented as an ESOCK plug-in. It implements all the protocols you'd expect – TCP, UDP, IPv4 and IPv6, and delegates the domain name system (DNS) functionality to a separate module – the Domain Name Daemon, which performs DNS lookups and caching for all applications on the platform. Despite this (internal) delegation, the interface to the DNS service is provided through the standard ESOCK APIs.

The IP stack has the ability to load plug-ins that can intercept and modify packets as they pass through the stack; however these interfaces are partner-only, and so are not described in this book.

Access to particular network technologies is performed using a combination of a network interface (NIF) and an agent (AGT). The normal separation of responsibility is that the network interface is a lightweight packet processor that sits on the data path, whereas the agent contains the logic to make and manage the connection. With certain network protocols though, the line is blurred, and therefore certain NIFs contain code to perform configuration of the interface – the Point-to-Point Protocol (PPP) NIF being a prime example.

On the control side, there is also the network interface manager (NIFMAN) and the network controller (NETCON), which, together with ESOCK, provide the framework for creating and managing IP connections.

NIFMAN also provides a system for plugging in network configuration mechanisms independently of underlying network technologies – at present the DHCP client implementation is the only example of such a plug-in.

## **SMS**

The short message service (SMS) stack is also a plug-in to the ESOCK framework. It provides various lower level services involved in sending an SMS, including fragmenting messages that are too long to fit in a single protocol data unit (PDU). It also provides APIs to allow applications to receive SMS messages before they are delivered to the messaging store – a useful feature for pushing information to your application without triggering an irrelevant indication to the user. These APIs are covered further in Chapter 8.

### **2.1.3 ETel**

ETel provides the lowest level interface to the telephony functionality within Symbian OS. Below ETel is the realm of the device-specific adaptation, which converts ETel API calls into whatever form is necessary for the specific baseband stack that is in use – this is the job of the TSY (Telephony SYstem module).

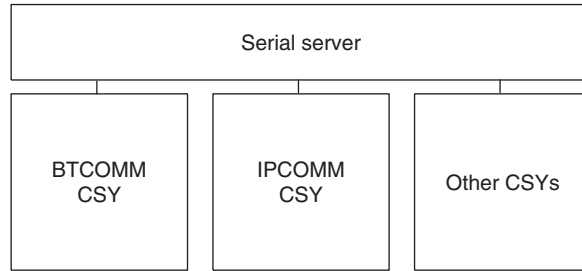
Because ETel provides a very low-level API, Symbian implement a wrapper over this called ‘ETel 3rd party’, which provides a limited subset of the functionality available in the ETel API in an easy-to-use form. A discussion of this API and the functionality available is in Chapter 7.

### **2.1.4 Serial Server**

The serial server provides, perhaps unsurprisingly, the interface to serial ports, both real and virtual. A series of plug-ins, called Comms SYstem plug-ins (CSYs, see Figure 2.4), provide access to specific technologies – RS-232 physical serial ports (rarely found on phones today), and Bluetooth, infrared and USB virtual serial ports. Some products contain additional, product-specific CSYs for communicating with other parts of the system, such as the cellular modem, but these are rarely useful to developers. Since USB is not covered in this book, and RS-232 ports are so rarely encountered, virtual serial ports over Bluetooth and infrared are covered in the respective technology chapters.

## **2.2 High-level Functionality**

Now it’s time to look at the functionality built on top of the low-level services that we’ve just discussed.



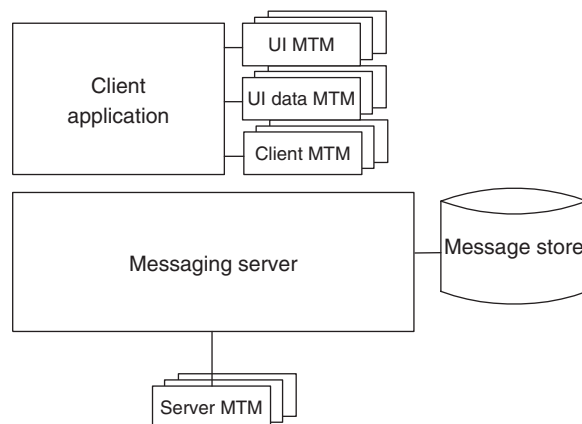
**Figure 2.4** Serial server and CSYs

## 2.2.1 Messaging

Messaging functionality in Symbian OS is based around the messaging server, and a series of plug-ins called Message Type Modules (MTMs), as shown in Figure 2.5. By default, Symbian OS-based phones ship with a rich set of MTMs built-in, including ones for SMS, MMS, IMAP4, POP3, SMTP and OBEX over Bluetooth and IrDA (for transferring objects between devices at short range). This set can be extended – there are MTMs available from many push email providers, as well as from other third parties.

A set of plug-ins to support a single messaging technology consists of four different MTMs – a server MTM, a client MTM, a UI MTM and a UI data MTM. Together they form a complete solution for adding a new message transport to the messaging architecture; although complex implementations may choose to provide some of the functionality of the UI MTM in separate applications. The purpose and responsibilities of each type of MTM are described in the second part of Chapter 9.

On top of messaging is built the SendAs service, which allows applications to send data to remote devices using an abstract API – this allows the set of technologies that can be used to send messages to be extended,



**Figure 2.5** Message server and MTMs

based on MTMs available on the system that support the SendAs service, without the application using the service having to change.

Chapter 8 covers receiving messages, and contains details of using various messaging APIs to access the message store. Chapter 9 covers sending messages using the SendAs service, including the UI-specific wrappers around it, as well as describing how to create a new set of MTMs to enable additional message transports to be added to the SendAs service.

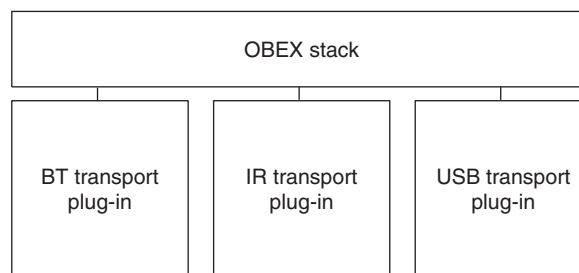
### 2.2.2 OBEX

OBEX was originally created by the IrDA specifications body as an efficient means of exchanging objects, and has since been extended to support a variety of other transport protocols, such as Bluetooth and USB. OBEX performs a similar role to the more popular HTTP protocol. There are many similarities between the two – both are request-response protocols, both are designed to transfer arbitrary data between devices, and both use the concept of headers describing the content and bodies containing it.

OBEX is designed to be easier to implement in resource-constrained devices than HTTP. It is also designed to be more efficient in terms of bandwidth, using a binary encoding scheme for headers rather than a text-based one. And the OBEX protocol itself is stateful, unlike HTTP which is fundamentally stateless and uses the additional concept of cookies to maintain state.

In Symbian OS, the OBEX stack is loaded into the client's process. As well as the core stack, there are a number of transport plug-ins as seen in Figure 2.6 – for IrDA, Bluetooth and USB. In order to use the OBEX stack the client's process must have the appropriate capabilities to perform the operations that OBEX attempts – the required capability is `LocalServices` for existing transports.

OBEX is used to transport data in a number of use cases – most commonly in the use case it was originally designed to fulfil – the transfer of objects such as vCards between devices. However, it is also used in a number of other Bluetooth specifications, including advanced image transfer and printing, and also as a transport protocol for OMA SyncML.



**Figure 2.6** The OBEX stack and transport plug-ins

Chapter 10 describes the Symbian OS implementation of OBEX, showing how to connect transport sessions, as well as build, parse and exchange objects.

### 2.2.3 HTTP

HTTP is most familiar as the protocol that serves up web pages on the Internet. Symbian OS contains an HTTP stack that is available for any application to use. The HTTP stack is loaded in the client's process, and can be extended with client-supplied plug-ins. As it runs in the client's process the client must have the appropriate capabilities to perform the operations that the HTTP stack attempts – currently this is the capability `NetworkServices`. Figure 2.7 shows the high level architecture of the HTTP stack.

As mentioned in the OBEX section, at a high-level HTTP and OBEX fulfil similar roles – they both provide a system for transferring arbitrary objects between a client and a server.

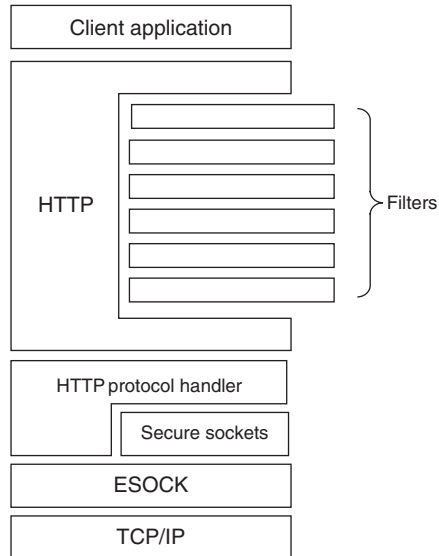
The Symbian OS HTTP stack supports HTTP/1.0 and HTTP/1.1, and as part of HTTP/1.1 it supports persistent connections and pipelining. It also supports basic and digest authentication, and HTTP over TLS using the 'https' URI scheme. A number of HTTP error codes, such as those for redirection are handled transparently. Chunked transfer encoding is supported, however, content encodings are not, by default, implemented, although that's not to say there isn't a filter providing them on a given UI platform.

The HTTP stack can be extended at runtime by the use of plug-ins called *filters*. These filters sit in the path between the client and the TCP socket and can listen for various events and modify ongoing transactions. The Symbian OS standard filter set includes a redirection filter, an authentication filter and a validation filter. UI platforms tend to introduce additional filters, including cookie handling and caching, some of which may get added to the default set of filters that are loaded when a client loads the HTTP stack. The HTTP stack is covered in more detail in Chapter 11.

### 2.2.4 OMA Device Management

With the increasing number of services available on mobile phones, one of the key technologies in making them easier to use is a method for remote management of configuration settings. For both network operators and medium and large companies, the ability to remotely provision and update settings on the user's device has the potential to greatly reduce the cost of deploying new solutions. This is the purpose of OMA Device Management.

Symbian OS contains a framework to which Device Management adapters (DM adapters) can be added to manage settings for specific applications using a generic device management protocol. This



**Figure 2.7** The HTTP stack and filter plug-ins

means application authors, by providing a DM adapter that exposes the configuration settings within their application in a standard form, can enable it for remote management.

The device management framework relies on the underlying SyncML framework to provide services common to device management and data synchronization. The common SyncML framework then provides a series of transport protocols over which it can be run. DM adapters plug into the DM framework, and provide standardized access to a variety of underlying settings stores – each settings store potentially being unique to a particular application.

## 2.3 Summary

That ends our overview of the different technologies described in this book. Hopefully you now have a high-level understanding of what each area does and how the different parts fit together. Next we'll start looking at each area in-depth. If there's a particular area you're interested in, feel free to skip straight to that chapter, although it might be worth consulting the reading guide in Chapter 1 to see if there is any suggested pre-reading for a given chapter.

