WILEY

Symbian OS Communications Programming

symbian

Michael J Jipping

EX-1016
U.S. Patent 8,825,801

symbian

Symbian OS Communications Programming

Jipping

WILEY

Symbian OS Communications Programming

Theresa LUNG-JIANG

Symbian OS Communications Programming

Michael J Jipping
Hope College,
Holland, Michigan USA

JOHN WILEY & SONS, LTD

Copyright © 2002 Symbian Ltd

Published by John Wiley & Sons, Ltd
Baffins Lane, Chichester,
West Sussex, PO 19 1UD, England

National 01243 779777
International (+44) 1243 779777

e-mail (for orders and customer service enquiries): cs-books@wiley.co.uk

Visit our Home Page on http://www.wileyeurope.com or http://www.wiley.com

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency, 90 Tottenham Court Road, London, W1P 9HE, UK, without the permission in writing of the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the publication.

Neither the authors nor John Wiley & Sons, Ltd accept any responsibility or liability for loss or damage occasioned to any person or property through using the material, instructions, methods or ideas contained herein, or acting or refraining from acting as a result of such use. The author and Publisher expressly disclaim all implied warranties, including merchantability of fitness for any particular purpose. There will be no duty on the author or Publisher to correct any errors or defects in the software.

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Ltd is aware of a claim, the product names appear in initial capital or capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration. 'The author' has asserted his moral right under the Copyright, Design and Patents Act 1988 to be identified as the author of this work.

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 0-470-84430-2

Typeset in 10/12pt Optima by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Biddles Ltd, Guildford and King's Lynn
This book is printed on acid-free paper responsibly manufactured from sustainable forestry, in which at least two trees are planted for each one used for paper production.

To my wife, Peg.
For her inspiration, encouragement, and patience ...but mostly for her patience

CONTENTS

Preface xiii

Foreword xvii

Acknowledgments xix

Section 1 The Basics of Symbian OS Communication 1

1 Introduction 3
1.1 Introducing Symbian OS 4
1.2 Symbian OS History 9
1.3 Symbian OS Reference Designs 11
1.4 Who Is this Book For? 14
1.5 An Overview of this Book 14

2 Introduction to the Symbian OS Communication Architecture 17
2.1 What Is a Communications Architecture? 18
2.2 Transport Technologies 19
2.3 Content Technologies 24
2.4 Requirements of Communicators and Smartphones 26
2.5 Implementation Techniques 28
2.6 Summary 30

3 Several Introductions to the Architecture Infrastructure 31
3.1 Basic Infrastructure Building Blocks 31
3.2 A Closer Look at the Infrastructure 33
3.3 Client-Server Relationships 37
3.4 The Communications Database 42
3.5 Infrastructure Issues 43
3.6 Summary 47

4 Transport Technologies 49
4.1 A Review of Communication Media 50
4.2 Serial Technologies 53
4.3 Network Technologies 63
4.4 Wireless Access Protocol 69
4.5 Bluetooth 70
4.6 Telephony 76
4.7 Summary 79

5 A Look at Content Technologies 83
5.1 Messaging 83
5.2 World Wide Web Technologies 97
5.3 Wireless Access Protocol 102
5.4 Other Content Technologies 105
5.5 Some Perspective on Content 105
5.6 Summary 106

6 Security and Communication 107
6.1 An Overview of Security Issues 107
6.2 Security Measures 114
6.3 Securing Transport Technologies 118
6.4 Securing Content Technologies 119
6.5 Summary 120

Section 2 Programming using Symbian OS Communication 121

7 Getting Started with Communications Programming 123
7.1 The Symbian Software Development Environment 124
7.2 Hello World! A Simple Example 127
7.3 Using the Emulators 133
7.4 Introducing TodoX: A More Extensive Example 136
7.5 Summary 139

8 Serial Communications 141
8.1 Programming Patterns for Communications 141
8.2 The Communications Server 148
8.3 The Serial API 149
8.4 Other Interface Calls 160
8.5 Active Objects 162
8.6 A Note About vCalendar Objects 167
8.7 Summary 168

9 Communicating via Infrared: Serial Communication 171
9.1 Admiring the Landscape 171
9.2 Serial Communication via the IR Port 173
9.3 vCalendar Objects 175
9.4 Summary 176

10 Using Sockets 177
10.1 Introduction to Sockets 178
10.2 Protocol Modules 178
10.3 RSocketServ: The Socket Server 180
10.4 RSocket: Using Sockets 181
10.5 Resolving Host Names and Addresses 191
10.6 Summary 194

11 Communicating via Infrared: Using Sockets 195
11.1 A Quick Protocol Review 195
11.2 System Definitions for IrDA 196
11.3 Initializing the Socket Server Connection 198
11.4 Opening the Socket 199
11.5 Configuring the Socket: Device Discovery 199
11.6 Exchanging Data 201
11.7 Terminating the Socket Connection 205
11.8 Using IAS 205
11.9 Using IrMUX 207
11.10 Using IrTranP 208
11.11 A Word About Expedited Socket Output 209
11.12 Summary 210

12 Communicating with TCP/IP 211
12.1 Considerations for Network Communication 211
12.2 The Network Interface Manager: NifMan 214
12.3 Initializing the System for TCP/IP 216
12.4 Opening and Configuring a TCP/IP Socket 217
12.5 Exchanging Data 222
12.6 Closing a Socket 222
12.7 Closing and Checking Network Interfaces 222
12.8 Another Example: Fetching a Web Page 223
12.9 Sending Datagrams 231
12.10 Summary 234

13 Bluetooth Communications 235
13.1 Considerations for Bluetooth Communication 235
13.2 Discovering Bluetooth Devices and Services 239
13.3 Communicating over Bluetooth 255

13.4 The Bluetooth Security Manager 259
13.5 An Example: Bluetooth Instant Messaging 262
13.6 Summary 274

14 Telephony 275
14.1 The Various Forms of Telephony 275
14.2 A Structural Overview 276
14.3 Components of the ETel Model 279
14.4 A Special Case Phone Call: Faxing 295
14.5 Summary 297

15 Sending and Receiving Messages 299
15.1 Overview of the Architecture 300
15.2 The Message Server 304
15.3 Accessing Messages 306
15.4 Creating Basic Messages: The `BaseMtm` Class 316
15.5 Easy Sending of Messages: Send-As Messaging 319
15.6 Receiving Messages 327
15.7 More Details on Specific Message Types 329
15.8 Summary 332

16 Browsable Content Technologies 333
16.1 HTTP and HTML 333
16.2 WAP 335
16.3 Summary 345

Section 3 Miscellaneous Topics 347

17 Synchronization: PLP and SyncML 349
17.1 Synchronization Mechanics 349
17.2 Some Early Attempts at Synchronization 352
17.3 PLP and Symbian Connect 353
17.4 SyncML 355
17.5 Summary 360

18 The Communications Database 361
18.1 Configuration and the Communications Database 361
18.2 The Communications Database API 362
18.3 Summary 377

19 Looking Ahead 379
19.1 The Next Step: GPRS 379
19.2 Internet Protocol version 6 382

19.3 Other Future Technologies 383
19.4 Summary 386

Appendix **387**

Index **391**

Preface

In a recent column in a trade periodical[1], a columnist named Brian Livingston declared that "the future belongs to devices we'll carry around, not boat anchors that must remain tied to our desks." I could not agree more. As I track the development of computers over time, it strikes me that some of the most innovative developments in computers have been done with handheld devices. These devices change the way we view and work with computing. In fact, the convergence of computing, communication, and mobility is one of the most significant innovations in recent years.

I have to agree again with Brian Livingston when he states "I believe the operating systems on the computer we'll soon be using—and taking for granted—will look like a newer, leaner OS named Symbian." Indeed, Symbian OS supports innovations with communication throughout its design. When a company like Nokia states, as it has, that 50% of its 3G devices will run Symbian OS by 2004, there is real recognition of this fact.

This book sets out to describe how Symbian OS supports the convergence of computing and communication. The designers of this operating system have built a strong communications architecture that can flex and adapt. This book covers how this architecture is built and how it can be used to take advantage of the communication capabilities of a device powered by Symbian OS.

Who Should Read This Book?

This book is written for the software developer who needs to design applications in the C++ programming language that harness communication on a Symbian OS device. This book assumes some familiarity with C++ in a Symbian OS environment.

If you are a programmer who needs to know how to write applications to send SMS messages, this book will show you how.

[1] Livingston, B., "The next Windows", *Info World*, 24:1 (2002), 30.

If you need to use C++ to access Internet services, this book will show you how. If you want to add infrared or Bluetooth capability to an application to exchange data with other devices, this book will show you how.

The Structure of This Book

This book is organized in three sections.

Section 1: The Basics of Symbian OS Communications

The first section introduces how communications work on Symbian OS.

Chapter 1 introduces the reader to Symbian OS: its structure, its history, and its implementation on various reference designs. It also gives an overview of the book.

Chapter 2 focuses on Symbian OS communications and introduces the Symbian OS communication architecture. The chapter looks at communications in two ways: transport and content. The technologies in each of these areas is discussed and related to support by the operating system. This chapter also discusses techniques used by Symbian OS to implement these technologies.

Chapter 3 looks at the system components that make up the infrastructure supporting communications. The designers of Symbian OS have developed several layered models to address the various aspects of communication and each is examined by this chapter.

Chapters 4 and 5 examine the support Symbian OS gives to transport and content technologies. In these chapters, we get into details about the technologies and discover what is necessary to support them.

Chapter 6 concludes this introductory section by examining issues of security that surround communications. Threats are examined from a Symbian device perspective and the security measures necessary to guard data are introduced.

Section 2: Programming using Symbian OS Communications

This section gets into the meat of writing applications that use the structure we presented in the previous section.

Chapter 7 introduces the programming environment that is used to develop Symbian OS applications. We present the traditional "Hello World!" application here. We also present an example

called "TodoX" that will be used in much of the rest of the section. This example application exchanges to-do list items between devices.

Chapter 8 covers the programming interface that is used to work with the serial port. Serial communication is probably the most familiar to programmers and is a good place to introduce the Symbian OS APIs. In this chapter, we introduce some important programming patterns that emerge as we work with communications devices. We also cover the concept of an active object, a special parallel thread of execution that deals with communications.

Chapter 9 is a rather short chapter dealing with serial communications over the infrared port. Its brevity is important: it emphasizes how little an application needs to be altered to move applications between use of the serial port and the infrared port.

Chapter 10 covers sockets, a concept that is used by Symbian OS on almost all non-serial communication. We go into much detail about socket properties and how a programmer would use the socket API. This chapter also covers the host resolver, an object that is used to discover properties of remote devices.

Chapter 11 revisits infrared communication using the socket concept. There are many ways to use the infrared port, and most are implemented using sockets. This chapter examines these new uses for the infrared port and serves as the first real example of socket use.

Chapter 12 examines the TCP/IP suite of network protocols. We start by reviewing TCP/IP and by examining the Symbian OS network manager. Then, using the patterns developed in previous chapters, we examine the many aspects of programming using TCP/IP. The chapter concludes with two examples: one that implements fetching Web pages from Web servers and one that implements a Network Time Protocol client.

Chapter 13 is devoted to programming using Bluetooth. The Bluetooth model is discussed and much space is given to how Symbian OS implements device and service discovery. Symbian OS uses sockets to communicate over Bluetooth, and this use of sockets is covered. The chapter concludes with a new example: sending instant messages over Bluetooth.

Chapter 14 provides a bridge between transport and content and covers telephony. Telephony provides strong evidence of how seriously Symbian OS takes the convergence of computing and communications. Telephony itself is introduced and the Symbian OS ETel programming model is discussed. The chapter discusses each of the ETel components and their APIs.

Chapter 15 switches from transport to content technologies and covers the Symbian OS support for messaging. Messaging is a crucial component in the functioning of communicators and smartphones. The Symbian OS model for messages is covered in detail, as is the API for using this model. The idea of "send-as messaging", which is an easy way to send any kind of message, is also covered.

Chapter 16 is devoted to the "browsable" content of HTML and WAP. The chapter covers the support given by the Symbian OS to HTML and its transport protocol HTTP. It covers in detail the layers in the WAP stack and how Symbian OS addresses them.

Section 3: Miscellaneous Topics

This section covers areas that are useful—but not crucial—to communications programming.

Chapter 17 provides an overview of synchronization issues with Symbian OS. It discusses general issues, and then focuses on two implementations of synchronization. Psion Link Protocol is discussed in detail and SyncML is introduced.

Chapter 18 covers the way applications move data into and out of the communications database. This database is used by Symbian OS to provide nonvolatile storage for communications parameters that would be used by more than a single application. The usage of the database and the API to access it are covered in detail.

Chapter 19 gives a look to the future and the technologies that will be addressed by Symbian OS in upcoming versions. Technologies like Multimedia Message Service and GPRS are coming soon; other technologies like 3G networking are a bit farther away. These future technologies are overviewed.

Foreword

David Wood, EVP, Partnering Evangelism & Research, Symbian

We live in times of tumult and change. In the midst of this excitement, two strong trends are clear. First, software is spreading more widely and more deeply into all aspects of society. Second, communication is becoming pervasive: more objects and gadgets talk to each other all the time, in ever-richer ways.

These two trends collide in the phenomenon of the mobile phone. The next generation of mobile phones uses incredibly sophisticated software to make it easy for their human owners to keep in touch, exchanging voice messages, text messages, pictures, email, and much more besides. These phones also constantly jump in and out of networks, with other phones and larger computers. As a result, they act as connected gaming devices, wicked entertainment devices, precocious pagers, personal and community organizers, mobile information portals, and as 'rich clients' for the forthcoming

waves of advanced data services and other compelling add-on software.

The software that lies at the heart of these new phones is Symbian OS. Endorsed by giants of the telecommunications industry—Fujitsu, Kenwood, Motorola, Nokia, Panasonic, Psion, Sanyo, Siemens and SonyEricsson—Symbian OS enables a dramatic leap in the power and utility of mobile phones. Symbian OS is neither a cut-down desktop/server OS nor an extended embedded OS, but a one-of-a-kind mobile operating system. Like all disruptive technologies, it has a challenging learning curve. Thankfully, it also has a flourishing community of software developers and trainers ready to help fellow travelers along this curve—sharing Symbian's motto of 'co-operate before competing'.

Symbian OS Communications Programming is a timely addition to the growing library of books about developing for Symbian OS. Dr. Michael Jipping brings a most welcome new perspective, carefully explaining how Symbian OS deals with communications of all sorts. I commend this book to you as a fascinating read, to help you towards creating your share of the huge market around Symbian OS phones.

Acknowledgments

The fact that you are reading these words means that my family—my wife and three kids—put up with me long enough to enable me to complete it. Their patience and perseverance—and excitement—helped put these words to paper.

Two former students—now computer science colleagues—deserve my thanks. Sarah Dieter and Joshua Krikke read the manuscript, corrected errors, examined the software, and let me know when I did not make sense.

Finally, I am grateful to the folks at Symbian who helped get this book out. In particular, Jonathan Hassell and Phil Northam were instrumental in this book's publication. Their encouragement and editorial skill are greatly appreciated.

Mike Jipping

Section 1

The Basics of Symbian OS Communication

1. Introduction pages 3–15
2. Introduction to the Symbian OS Communication Architecture pages 17–30
3. Several Introductions to the Architecture Infrastructure pages 31–47
4. Transport Technologies pages 49–81
5. A Look at Content Technologies pages 83–106
6. Security and Communication pages 107–120

1

Introduction

On February 14, 1946, a group of scientists and engineers activated the world's first general-purpose computer. The Electronic Numerical Integrator and Computer—ENIAC—was a huge computer that took up the equivalent of a small office building. It was built in 30 separate units and used 15,000 square feet of floor space to house 17,480 vacuum tubes and 1,500 relays. At a weight of 30 tons, it consumed 140 kilowatts of power. Its sole purpose was to compute tables of numbers; in its day, it served a crucial purpose by producing trajectory computations for field artillery faster and more precisely than humans could produce the same computations. It could actually compute 5000 additions per second.

Today, computers have evolved from 30-ton behemoths into much smaller machines that can do far more than simple number crunching. Handheld devices and mobile phones are at the leading edge of this evolution. With these devices we are seeing the convergence of communication, computing and mobility.

A Nokia 9210 Communicator is a good example of this convergence. This handheld device combines an ARM 9 processor with 32 MB of memory and digital GSM (900/1800/1900 MHz) communication capabilities in a slim mobile phone size. It can execute over 220 million instructions per second, help manage your personal life, let you make mobile telephone calls, and show videos on a screen that can handle over 32,000 colors. It harnesses the power of a desktop workstation and can fit into your pocket.

The Nokia 9210 is not only a good example of the advanced capabilities that modern computers have, but it is also a good example of the multiple purposes for which such computers are built. While the first computers were designed with one purpose—to calculate numbers quickly and precisely—today's devices have many purposes. They entertain by playing music or games. They manage information by storing files and displaying data. They schedule your

day and remind you of appointments. They communicate with each other and the outside world.

In fact, as computers and their operating systems continue to grow and develop, the convergence of this computing power with the ability to communicate becomes an important element that lies at the core of what a computing device does. Along with other facets of computing (such as processor speed and display capability), communication capability is a deciding factor for many people, determining what computer they buy and use. Some current technologies that are key to computer use are messaging—via email or faxing—and browsing the World Wide Web.

In the future, communications will become even more important. This means device-to-device communication, the ability to use the Internet, making phone calls, sending all kinds of messages—email, fax, or SMS (short message service)—and browsing the World Wide Web. The future will bring easy access to information that is stored anywhere in the world, using the Internet as a storage facility. Future communications will feature mobile, wireless connections as well as conventional wired access, and will connect people to each other by way of the devices they carry.

The focus of this book is communication—specifically, on communication using handheld devices. More to the point, this book presents an operational and programming view of communications on handheld devices using Symbian OS. We present Symbian OS in this book and how it can be used to harness the communication power of handheld machines. We will explore the concepts that drive the way Symbian OS implements communication. We will spend a large part of the book examining the interfaces that are used to program communication facilities.

This chapter is an introduction to the book. We will look closely at Symbian OS and at its history; we will examine how Symbian OS is applied to various handheld device designs; and we will conclude with an overview of the rest of the book.

1.1 Introducing Symbian OS

Symbian OS is a *system*: a combination of several different elements that can be applied to several different designs of computing devices. There are six elements that combine to make up Symbian OS:

- the core operating system, commonly called a *kernel*
- a collection of middleware for system services
- a set of resource managers, called *application engines*

- a framework for designing user interfaces
- methods for synchronization with other machines
- a Java virtual machine implementation.

These elements combine to produce software made to control devices from handheld computers to mobile phones. We will overview each element on its own.

Web Sites to Check Out

You can do a lot of research for yourself at Symbian's Web site. There are many white papers on Symbian OS and a large number of practical examples to introduce developers to the Symbian community. The URL is ***http://www.symbian.com***.

The Symbian Developer Network is Symbian's developer organization. This site supports developers of software for Symbian OS: it has Symbian OS software development kits, hosts forums for questions and answers, and contains patches and examples of software designs. You will find Symbian DevNet at ***http://www.symbian.com/developer***.

Both of these sites provide a growing library of excellent reference material as you develop software for Symbian OS.

1.1.1 The Symbian OS Kernel

The heart of Symbian OS is the core operating system: a collection of device drivers, data tables, and programs that allows the user to work with a computer's hardware. This is the common core of Symbian OS, present in all Symbian OS devices.

Symbian OS is a *kernel-based* operating system. There is one program that runs at all times and manages the services that are provided to the user. This program and its data tables—collectively called the *kernel*—need to be small and efficient. Only the elements of computer operation that need constant attention are placed in the kernel; other functions are moved out to middleware elements or applications. This design makes the kernel very compact and makes the architecture and operation of Symbian OS very modular.

From its beginnings, when it was known as the EPOC operating system, Symbian OS has had an object-oriented design and an efficient, tight structure. It is a 32-bit operating system that supports multitasking and multithreading. Its modular structure supports and encourages a wide variety of communication components and supports the ability to add to and extend core communication

elements to adapt to new devices and methods. The communications architecture derives great benefits from this modular design, as we will see.

Symbian OS has seen several releases. The most recent is Symbian OS version 7.0. New devices, such as the Nokia 7650, are using this version to support new technology such as GPRS and Bluetooth.

1.1.2 Middleware

The term *middleware* refers to the collection of elements—libraries, data storage, and programs—that implement system services, but do not need to be in the kernel. In Symbian OS, this includes implementations of services such as data management, communications, and graphics. As an example of middleware, consider the window system in Symbian OS. The window system forms an essential part of how a user interacts with the computer, but management of windows and stylus taps is not essential enough to be built into the kernel. By contrast, management of the display device itself (i.e., the server) is in the kernel. The transformation of a display device into a set of windows is left to middleware.

Symbian OS uses *servers* to implement middleware. The idea it uses is that a server can manage a particular service by accepting requests from many different sources—or *clients*—and coordinating access and implementation by its response to those requests. To follow our example, the window system is not built into the Symbian OS kernel, but rather implemented by a *window server* that receives requests from applications and manages the display. These servers lie in the middle area between the kernel and the user—hence the name *middleware*—and they contribute to the modular design of Symbian OS. By creating a new layer for middleware, Symbian OS designers made it easy to design new system services and to upgrade existing services without rewriting the core of the operating system.

Middleware servers exist for many system services, including the window system, network communication, serial and infrared implementations, multimedia management and database implementation.

1.1.3 Application Engines

User level applications on Symbian OS also benefit from the kind of coordination that middleware provides. Coordination of access to non-essential resources is done through *application engines*.

These form single access points for user level applications as they access resources. Having a single object that coordinates access to an Agenda database, for example, saves memory and processing time by channeling all manipulation of that database through one implementation.

Like middleware, application engines typically take the form of servers. The difference, however, comes from the area being managed. Application engines manage *application* data and services, not *system-oriented* data and services. For example, while Agenda data is certainly crucial to a user for managing her schedule or to-do lists, it is not crucial to the management of the computer. In fact, it is likely that application engines will themselves interact with middleware servers.

The application engines included in the Symbian OS core are:

- the Agenda engine
- the Contacts engine
- the Sheet engine
- the Alarm server and World Time engine
- the Spell engine
- the Help engine

1.1.4 User Interface Framework

Since computers that run Symbian OS are typically small handheld devices that are specifically oriented toward users, the user interface is extremely important. It is important that this interface be easy to use, easy to change, and easy to program. In addition, there are several different device designs that are meant to run the Symbian OS, so this user interface must be adaptable. In the face of these constraints, the Symbian OS designers chose to include a *framework* on which to build user interfaces as part of the core operating system.

By choosing to include a graphical user interface framework rather than one specific graphical user interface, the designers of Symbian OS laid a foundation upon which many different user interfaces can be built. In the current version of Symbian OS, there are two user interfaces that build on the GUI framework. These interfaces use common GUI components, such as controls and dialogs, but the specific look-and-feel of these components is left in the hands of the mobile device manufacturer licensing the Symbian OS so that they control each specific device implementation.

Currently, the core GUI components are Uikon and Standard Eikon. The core libraries of Uikon contain elements that are largely

common across all reference designs. Standard Eikon contains starting-point code for modules that occur in most reference designs, but whose functionality typically needs to be altered. A "control factory" element allows the GUI framework to be extended. The intent of this type of structure is to allow the drawing of core and additional controls in such a way that a device manufacturer can change the appearance of the GUI component without affecting the drawing code in individual controls.

1.1.5 Synchronization Technology

In addition to the core technology of Symbian OS, the platform also includes ways to synchronize data with other computers. This type of technology has been the subject of industry-wide cooperation and standards development and Symbian has integrated these developments into the Symbian OS. On Symbian OS, synchronization is implemented in three discrete parts.

- The *connection manager* is a user-initiated process that runs on a Symbian OS Device and detects when another computer wants to connect. It starts a synchronization connection when such an attempt is detected.
- *Connectivity servers* implement the various functions of a synchronization session: simple file browsing, file synchronization, backup, or restoration.
- *File converters* transfer data between formats specific to certain applications. These converters are minimal on the Symbian OS side of the connection; most conversion will probably happen on the connecting computer. However, converters do exist to convert rich text format to HTML and some office-type Symbian OS applications to their Windows counterparts.

Synchronizing is done via a "conversation" between the connectivity servers on two computers. That conversation takes place via a specific protocol or language. Historically, on machines based on Symbian OS, that protocol has been Psion Link Protocol (PLP). This proprietary protocol was invented by Psion Computers and requires a special interface on a PC, which is supplied by Symbian. Symbian OS also uses a more recently developed, and more widely adopted, protocol called SyncML. This protocol was developed in 1999 as a standard for handheld machine synchronization.

1.1.6 Java Virtual Machine Implementation

Symbian OS includes a full Java implementation within what is called the "J2ME" framework. The Java 2 Micro Edition framework

is an implementation of the Java Virtual Machine (JVM) as specified by Sun Microsystems for small handheld devices. While this JVM lacks some services and interfaces of the Java 2 Standard Edition, it does include some Java technologies that are tailored specifically for devices that adhere to the Micro Edition specification. Among these are a JavaPhone implementation and an extensive PersonalJava implementation. Coverage of this part of Symbian OS is beyond the scope of this book, and the reader is referred to the many published works of Sun Microsystems, the Java Community, and Symbian to look into this part of the Symbian platform further.

Further Reading on Java

We will not be covering Java in this book. If you want to know more about programming in Java on Symbian OS, I would refer you to a book by Jonathan Allin called *Wireless Java for Symbian Devices*, published by Symbian Press through Wiley. It provides a great reference to using Java on the Symbian platform. You can find out more details at ***http://www.symbian.com/books***.

1.2 Symbian OS History

The heritage of Symbian OS begins with some of the first handheld devices. The operating system began its existence in 1988 as SIBO, an acronym for "sixteen bit organizer". SIBO ran on computers developed by Psion Computers, which developed the operating system to run on small-footprint devices. The first computer to use SIBO, the MC laptop machine, died when it was barely out of the gate, but several successful computer models followed the MC. In 1991, Psion produced the Series 3: a small computer with a half VGA size screen that could fit into a pocket. The Series 3 was followed by the Series 3c in 1996, with additional infrared capability; the Sienna in 1996, which used a smaller screen and had more of an "organizer" feel; and the Series 3mx in 1998, with a faster processor. Each of these SIBO machines was a great success, primarily for three reasons: SIBO had good power management, included light and effective applications, and interoperated easily with other computers, including PCs and other handheld devices. SIBO was also accessible by developers: programming was based in C, had an object-oriented design, and employed application engines, a signature part of Symbian OS development. This engine approach was a powerful feature of SIBO; it made it possible to

standardize an API and to abstract formats away from the application programmer.

In the mid-1990s, Psion started work on a new operating system. This was to be a 32-bit system that supported pointing devices on a touch screen, used multimedia, was more communication-rich, was more object-oriented, and was portable to different architectures and device designs. The result of Psion's effort was the introduction of EPOC Release 1. Psion built on its experience with SIBO and produced a completely new operating system. It started with many of the foundational features that set SIBO apart and built up from there.

EPOC was programmed in C++ and was designed to be object-oriented from the ground up. It used the engine approach pioneered by SIBO and expanded this design idea into a series of servers that coordinated access to system services and peripheral devices. EPOC expanded the communication possibilities, opened up the OS to multimedia, introduced new platforms for interface items like touch screens, and generalized the hardware interface. EPOC was further developed into two more releases: EPOC Release 3 (ER3) and EPOC Release 5 (ER5). These ran on new platforms like the Psion Series 5 and Series 7 computers.

As EPOC was being developed, Psion was also looking to emphasize the ways that its operating system could be adapted to other hardware platforms. From mobile phones to Internet appliances, many devices could work well with EPOC. The most exciting opportunities were in the mobile phone business, where manufacturers were already searching for a new, advanced, extensible and standard operating system for the next generation of devices. To take advantage of these opportunities, Psion and the leaders in the mobile phone industry—for example, Nokia, Ericsson, Motorola, and Matsushita (Panasonic)—formed a joint venture, called Symbian, which was to take ownership of and further develop the EPOC operating system core, now called Symbian OS.

Symbian OS was explicitly targeted to several different generalized platforms. It was flexible enough to meet the industry's requirements for developing a variety of advanced mobile devices and phones, while allowing manufacturers the opportunity to differentiate their products. It was also decided that Symbian OS would actively adopt current, state-of-the-art key technologies as they became available. This decision reinforced the design choices of object-orientation and a client-server architecture.

1.3 Symbian OS Reference Designs

Symbian OS is intended for a range of devices. To assist in the design of devices that can use the capabilities of Symbian OS, Symbian has developed a series of reference designs for mobile devices and phones.

Keyboard-based mobile phones are device designs that are generally referred to as *communicators*—information devices with communication capabilities. This design most closely resembles the historical line of devices that have used EPOC and Symbian OS. It is a design that reinforces the state of the art: half VGA color screen with a keyboard and a rich application suite derived from previous implementations. Figure 1.1 shows a Symbian concept device and Figure 1.2 shows a Nokia 9210. Both devices feature keyboard navigation, 16-bit color screens, sound, and a rich communication suite. As an example of design customization, the generic concept device specifies a touch screen interface while the Nokia 9210 does not use a touch screen at all. Instead, it implements user interaction through buttons on the right of the display screen.

Single-handed use mobile phones are devices that are typically designed with advanced data capabilities. These are categorized as

Figure 1.1 A Symbian concept communicator design.

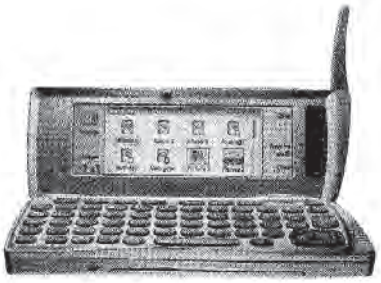

Figure 1.2 The Nokia 9210 communicator.

smartphones—communication devices with information capabilities. This design puts browsing content, messaging, and access to entertainment in the forefront and other information management as a secondary goal. A Symbian concept device for a smartphone is shown in Figure 1.3 and a Nokia smartphone—the 7650 phone—is shown in Figure 1.4. Each design includes the integration of a color touch-screen with a communications device. These devices can function as mobile telephones or wireless network nodes. The Nokia 7650 includes technologies like GPRS, Bluetooth connectivity and digital imaging (via a digital camera).

Pen-based mobile phones are communicators without a keyboard. They have a tablet-style form factor that allows additional data inputs, such as a large touch screen (which differentiates it from

Figure 1.3 A Symbian smartphone concept design.

Figure 1.4 The Nokia 7650 phone.

the smartphone above). The Symbian "generic" concept device is in Figure 1.5. An Ericsson concept version is shown in Figure 1.6. This design is built around a quarter VGA sized screen with no keyboard. The device uses handwriting recognition. It is oriented around a browser style interface, and operation of the unit is done completely through a touch screen. This category of device includes the Ericsson R380, the first Symbian OS phone.

Symbian OS allows mobile phone manufacturers to customize the operating system while maintaining an industry-standard common core for all devices. The real-world reference designs

Figure 1.5 The Symbian pen-based mobile phone concept design.

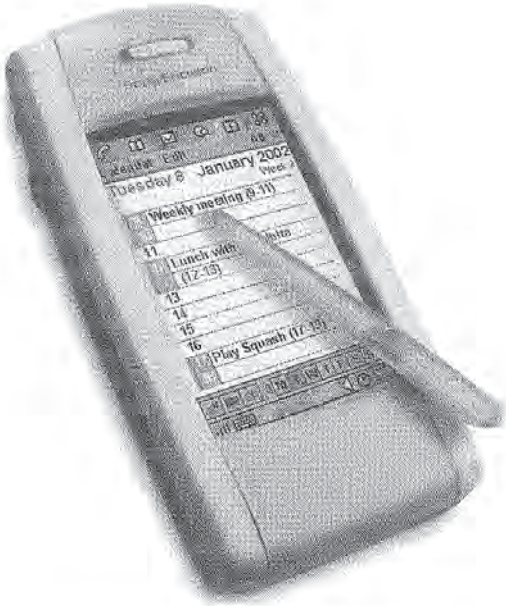

Figure 1.6 The Sony Ericsson P800.

described above derive from generic reference design work done by Symbian. The mobile phone categories above reflect generic Symbian OS reference designs that have been used by Symbian and its licensees in the past.

There are many common components to these device designs and communication plays a strong central role in each. As is the duty of operating systems, Symbian OS provides access to system services; I will focus in this book on the access that is provided to communications. Each reference design provides a different way for the user to access communication methods and Symbian OS integrates different ways of communication. Programming interfaces and the technologies are basically the same across the board. The communications architecture is standard and that is what we will be studying here.

1.4 Who Is this Book For?

The content of this book is targeted at software developers who know something about programming and are looking to expand their knowledge set. The programming in this book will be in the language C++ and will use the software development kits (SDKs) provided by Symbian and its partners.

While software can be designed using other means, the Symbian SDKs are meant to be used with Microsoft Visual C++. Knowledge of this development environment is very helpful.

A Companion Book

There are many good books that can serve as prerequisites or companions to this book. One that I highly recommend as a precursor to tackling this material is *Professional Symbian Programming*, by Martin Tasker, *et al.*, published by Wrox. It provides an excellent introduction to programming using the Symbian platform APIs and is a good way to get started on Symbian programming.

1.5 An Overview of this Book

Aside from this chapter, this book's primary focus is on communication in the Symbian platform. We will cover the communications architecture in three major sections.

- **Communication Basics:** Chapters 2 through 6 will cover the basic concepts and methods used in the Symbian platform's approach to communications. We will introduce the architecture, and then look closely at its infrastructure. We will then go over transport and content technologies in detail, and wrap up with a look at communication security and how it plays in integral role in Symbian platform design.
- **Programming Using the Communications Architecture:** Application programming interfaces (APIs) are the major way that the goals and criteria set up by Symbian are expressed in the design of the communication architecture. Therefore, our examination of APIs and their use will occupy a large part of this book. Chapters 7 through 16 will work through the communication architecture and show how to program using its structures. We will spend time on each of the technology areas—transport and content—outlined above.
- **Miscellaneous Topics:** Before we leave the communications architecture, there are some miscellaneous concepts that are implemented by Symbian that warrant some attention. These include data and device synchronization standards, accessing the communications configuration database built into Symbian OS, and future directions for the platform. Chapters 17 through 19 will overview these topics.

Accompanying Software

All code examples that we will work through in Chapters 7 through 18 can be found on the Symbian Web site at ***http://www.symbian.com/books/socp/socp-info.html***. In addition, throughout this book I will occasionally suggest using software to review some aspects of Symbian OS or its components. You will also find any software development kits from Symbian that you need to get started programming on your own (see Chapter 7). All of this software will be available from this Web site.

I have found examples to be my best resource when learning new programming systems. I have attempted to include as many examples as I could that would illustrate the features we discuss here. I have also included some public domain software and their sources. You will find installation instructions and some tips on using the software at the support site.

2

Introduction to the Symbian OS Communication Architecture

In his book, *Clicks and Mortar*, David Pottruck analyzes the culture of computers and the Internet and applies this analysis to business practices. As one of his observations, he notes that this culture of computers and information we are in is not really the *information age* as it commonly called, but rather it is the *communications age*. Our times do not really focus on information but on the communication that enables us to share and use that information.

Symbian OS reflects this kind of relationship between information and communication. While Symbian devices are indeed information devices, much of the focus of Symbian OS is on a communication-rich environment that is flexible enough to support many forms of connectivity. The organization of the communication layers of Symbian OS should be viewed as more than simply infrastructure; it is in fact an *architecture*: a combination of concept, design, structure, and function.

A majority of this book will be dedicated to the structure and function of Symbian OS communications architecture in the form of tutorials and programming references. This chapter, however, introduces the basics of the communications architecture: the concept and design behind it. In this chapter, we will overview the concepts that comprise the architecture; in the next chapter, we will overview the design that the architecture implements.

Symbian OS was designed for a certain type of computing device; I introduced these devices in Chapter 1. From communicators to smartphones, these devices can vary widely in shape, size, and even function. They all have common needs that are addressed by the various facets of Symbian OS. In this chapter, we will discuss Symbian OS design by outlining the communication requirements of these devices. We will then proceed to discuss the ways that Symbian OS

meets those requirements: through a client-server design, through the use of a communications database, and through the use of synchronization. We will complete this chapter by looking at issues that surround the security of information and communication.

2.1 What Is a Communications Architecture?

Architecture is a term used to describe the way that form and function implement design. A building, for example, has an architecture—that architecture is displayed as the way that bricks, girders, and wiring work together to reflect a design that was meant to service the building's inhabitants and the designer's artistic sense. To many, architecture is little more than the shape of the windows or the balance of bricks to glass. But to the architect, there is a set of layers, a collection of facets, a sequence of details for which the final structure is simply the last step.

In the same way, the various communication components of Symbian OS unite together. There is more than simply infrastructure in Symbian OS's communication architecture. It is more than an implementation; it is the combination of concept and design, structure and function into a single expression. There are concepts embedded in the way the pieces work and fit together. There is a design to each component; a structure and a function to each piece. Together, the components form a communications architecture that can serve the needs of each device design and each user in a well-designed and cohesive manner.

The communications architecture is, then, an expression of a specific set of design goals and criteria.

- *The communications architecture must support all user communications applications.* Users of smartphones and communicators expect a level of functionality from their devices. The communications architecture must support this functionality and be able to address future needs gracefully.
- *Communications components must be exceptionally flexible to cope with the mobility designed into these device platforms.* Users of communicators and smartphones are mobile and demand a mix of connectivity methods and communication platforms. The communications architecture must easily adapt itself to the changing requirements of configuration and connectivity. On-the-fly reconfiguration, for example, from a dial-up connection to a wireless network must be easy and straightforward.

- *Communication components must be organized to accommodate the constant restructuring and rebuilding of communication technology.* The communications architecture must be built in a modular fashion so that pieces can be replaced as technology evolves without upsetting the entire structure. In fact, the structure should be able to accommodate the coexisting of old and new pieces.
- *The communication components must adapt to the tight RAM and CPU constraints of their intended target platforms.* While they must do a great amount of work, the components of the communications architecture must not consume a burdensome amount of resources. The resources of a communication device are to be targeted at an application, not consumed by communication methods.

This is a difficult job: support all functionality possible in a flexible, modular fashion in a limited computing environment! Despite the daunting task, the designers at Symbian believe they have achieved an architecture that meets these criteria.

The design of the communications architecture addresses these issues in two parts: *transport* technologies and *content* technologies. Transport technologies address the way data is communicated between devices. Content technologies address the data that is carried by a transport method. For example, in the case where someone is browsing a Web page, the HTML content of the page has been transported to the viewing application over a wired network using TCP protocols.

2.2 Transport Technologies

Transport technologies is an area unseen by most users of communicators or smartphones. It involves *communication media,* or the physical components that convey the signals between devices, and *communication protocols,* which is what we call the structure of the conversation between devices that allows them to move data between them.

There are many transport technologies and Symbian OS addresses several of them. I group these technologies into *point-to-point* technologies, where two devices communicate directly with each other, and *networking* technologies, where the communication medium is shared and devices have the potential to communicate with several other devices over this medium.

2.2.1 Point-to-Point Transport Methods

Serial and infrared communication are the two most widely used point-to-point methods.

Serial transport is one of the oldest communication methods for computers. This transport method typically runs over short distance cables, which conform to one of two standards: RS-232 or RS-432. These cables are comprised of wires in a 9 or 25 wire group. Despite the many wires, only two of them are used to communicate data, one bit at a time, in sequential order, back and forth. One wire transmits data in each direction. The other wires are used as control pins, carrying signals that convey information about traffic flow and data management.

There are liberal interpretations of RS-232 standards in practical implementations. For example, some connections use only three wires: two for data and a signal ground. This three-wire, stripped-down version is still referred to as an RS-232 connection.

Most point-to-point communication methods exchange data between devices in one of three ways:

- *Simplex communication* exchanges data in one direction only. Using this method, one device is a sender and one is a receiver—they never switch roles. Since communication is one-way, the sender never knows if the receiver actually got the information sent and if that information was sent without errors.
- *Half-duplex communication* exchanges data in two directions—both sides can be a sender and a receiver—but data flows in one direction at a time. In half-duplex modes, a single wire can be used to exchange all the data, because the devices involved take turns sending and receiving.
- *Full-duplex communication* is also a bi-directional method, but data can be exchanged between devices in both directions at the same time.

RS-232 communication is full-duplex, using two wires to send data from both sides.

Serial communication is typically controlled on a computing device by a Universal Asynchronous Receiver Transmitter (UART) chip. Currently, UARTs are capable of driving serial transport up to speeds of 115,000 bits per second (115 Kbps).

The infrared (IR) transport method uses light in the infrared spectrum to convey data between two devices. Because it is based on light, IR communication is limited to devices that can directly "see" one another and are within a short, limited range.

There are many standards that infrared communication uses; all of them fall under the umbrella of the Infrared Data Association (IrDA). IrDA specifies that data signaling is bi-directional (however, only one side can transmit at a time) and can operate in speeds from 9600 bps to 4 Mbps. Symbian OS supports IrDA standards that include a serial port emulation protocol (IrCOMM), a multilayered protocol that supports device discovery and data segmentation (IrLMP and Tiny TP), a protocol to exchange digital pictures between devices (IrTranP), and a protocol stack that allows access to wired networks (IrLAN). These standards will be more fully explained in later chapters.

2.2.2 Networking Technologies

To facilitate faster communication, networking technologies address situations when many devices communicate over the same communication medium. Because of their complexity, network implementations are characterized in protocol layers or stacks. The official stack structure (specified by the International Standards Organization (ISO)) is shown in Figure 2.1.

Networks are characterized by *datagram delivery* of data, that is, the delivery of data in the form of small packets that are broken

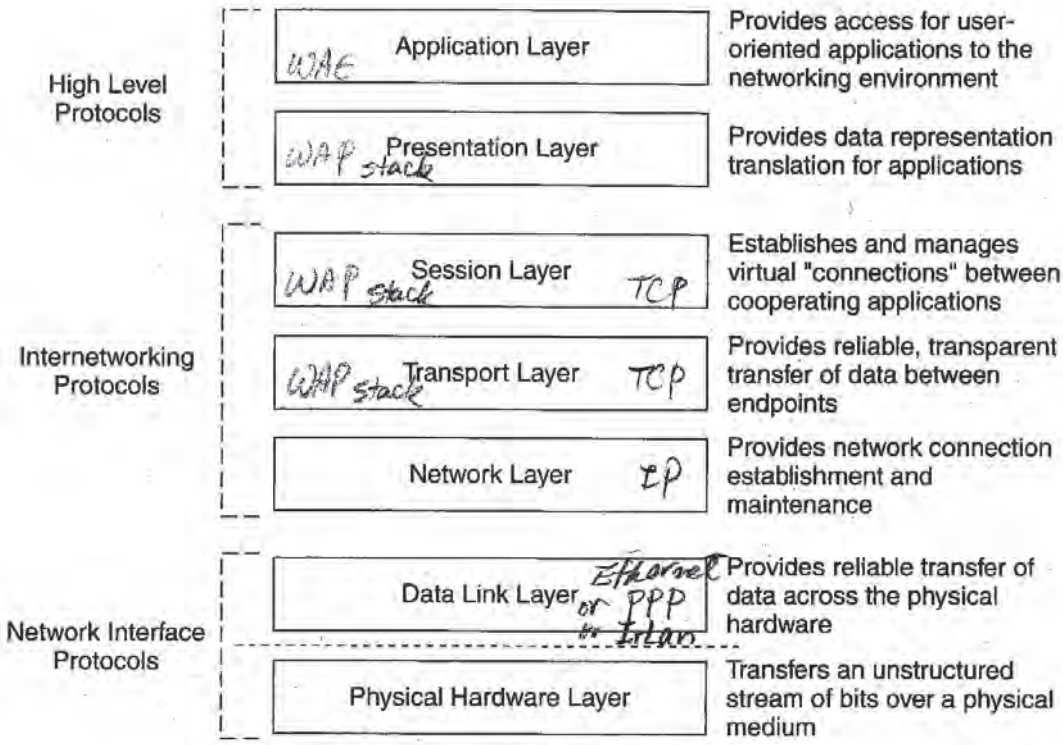

Figure 2.1 ISO protocol stack.

up, passed over the transport medium, and reassembled at their destination. Network data is digital data, and a digital medium is needed to pass that data. While analog methods may be used for transmission (e.g., serial cables or IR), the physical layer will translate the analog data to digital data.

Symbian OS includes support for several networking technologies:

- *Ethernet:* This is by far the most widely implemented and most popular form of networking. Ethernet is a passive form of networking that shares datagrams among all machines that are on the network. It relies on each machine to be well-behaved and to pay attention to only the packets that "belong" to it. Conventional Ethernet technology runs these networks up to 100 Mbps, although most installations of Ethernet are still at 10 Mbps. The newest implementations of Ethernet are currently running at 1 Gbps. Ethernet is a data link layer technology.
- *Point-to-point protocol:* Point-to-point protocol (PPP) is a way of implementing networking protocols between only two machines instead of a multiple machine environment. Networking technologies are still supported, as PPP operates at the data link layer. PPP is often used to allow a device that uses a telephone connection to graft itself into a larger network (e.g., the Internet) via a special "access server" computer that acts as a bridge between the PPP source and the destination network.
- *TCP/IP:* Transmission Control Protocol/Internet Protocol (TCP/IP) is a very widely implemented and stable networking protocol suite. TCP/IP implements the session and transport layers through TCP, and the network layer through IP. The suite is most often implemented over Ethernet or PPP, using these protocols to pass actual datagrams. TCP/IP implements addressing of devices and routing between devices and encompasses a large collection of protocols. Symbian OS includes an implementation of the TCP/IP core protocols.
- *WAP:* The Wireless Access Protocol (WAP) suite is a set of protocols that implements much of the ISO stack but is tailored to small devices such as mobile phones and other handheld devices. The Wireless Application Environment (WAE) implements the application layer, but the WAP stack implements the presentation, session, and transport layers, and includes an additional layer of security on top of the transport layer. The

WAP stack leaves the network, data link and physical layers to wireless transport providers ("bearers" in WAP terminology). Symbian OS has full support for the WAP stack.

The IETF

Internet protocols and access procedures are the purview of the Internet Engineering Task Force (IETF). This nonprofit organization exists to organize the efforts to extend the Internet, specifically in the area of its protocols and security. In addition to meeting regularly to discuss these issues, the IETF issues a collection of Request for Comment (RFC) documents. These RFCs are the definition documents for protocols; the collection forms the basis for the open standards nature of the Internet.

More information on the IETF and their RFC collection can be found at ***http://www.ietf.org***.

2.2.3 Bridge Technologies

There are technologies that bridge the difference between point-to-point and networking technologies. These are technologies that can be used in either setting. Symbian OS supports two of these: Bluetooth and GSM.

Named for a 10th century Viking king, Bluetooth is a short-range, radio-based, wireless protocol suite that is meant to facilitate low power, close-range communication. It was developed as a point-to-point connection strategy that uses the 2.4 GHz radio frequency and allows communication between devices up to 10 meters apart with a gross data rate of 1 Mbps. Much of Bluetooth's appeal comes from its use as a replacement for data cables. Individual point-to-point connections can be tied together to form a personal area network, called a piconet. The Bluetooth specification describes its own protocol stack that includes security protocols as well as those meant for layers in the ISO layer model. The heart of Bluetooth is realtime device discovery and interactive implementation. The Symbian OS implements the Bluetooth stack.

The Global System for Mobile Communications (GSM) is a mobile telephony implementation. It is used by digital mobile devices to exchange information over long or short distances in a wireless fashion. GSM is mostly implemented for mobile phones, and for devices that incorporate mobile telephony as well as voice/data traffic. A central feature of GSM is a messaging service, detailed in the content technology section below. Symbian OS includes support for GSM communication.

2.3 Content Technologies

For the user, it is the content of the communication that matters, not the way it is transported. Thus, the proper implementation of content technologies is very important. Symbian OS addresses several important technologies.

2.3.1 Messaging

The convergence of computing and communication in communicators and smartphones presents some interesting opportunities. One of these is the use of *messaging*: the two-way exchange of various types of messages between devices. Messaging presents several challenges; one of the largest is the sheer variety of the types and technologies of messages currently exchanged between computers. Symbian OS pulls the disparate technologies together and presents a consistent programming interface to the software developer. In addition, Symbian OS is designed to be extensible, i.e., it can incorporate new forms of messaging as they are invented. Among the types of messages that Symbian OS currently supports are these:

- *Email messaging:* One of the most common and widely used forms of messaging is electronic mail. This form was originally invented as an electronic method for sending text-based messages—"mail" for human consumption—between computers. Each message has an "envelope" and a message "body". The envelope contains information about the message—sender, intended recipient, date and subject of message, etc.—in the form of a message header. The message body is typically a textual communication between people, but it can also contain objects, textual or binary data, encoded in special ways so that they can be represented in a text-based message.
- *SMS messaging:* SMS messaging is a form of messaging specific to GSM-based communication. SMS—short for *Short Message System*—uses text messages of 160 characters or less for communication between devices on GSM networks. While textual in nature, these messages can convey special data to devices; specialized data such as graphics or phone ringtones can be encoded in an SMS message.
- *BIO messaging:* BIO messages—short for *bearer independent object*—are messages that are intended for the *system* rather than the *user*. Sending a graphic image contained in a message to a phone for configuration purposes is a good example of BIO messaging. This type of messaging includes configuration

information (e.g., system configuration or browser configuration), specially formatted contact and scheduling information (e.g., vCard and vCalendar), and email notification.

- *Fax messaging:* A fax—short for *facsimile*—message is a form of message that contains a digitized image, organized into pages, intended to also convey information about physical paper pages. The support in Symbian OS for faxing includes various methods of fax encoding, multiple recipient outgoing faxes, sending and receiving both standard and fine resolution fax images, and the unattended receipt of fax messages.
- *Other forms of messaging:* Obviously, the designers of Symbian OS could not accommodate all forms of messaging that exist or predict the forms of messaging that will be invented. Therefore, there is also a mechanism built into the platform that can accommodate other messaging forms not designed into it. *Message Type Modules* (MTMs) are implemented by a collection of objects that can be extended to implement any form of messaging that might be needed. An MTM features the receipt, processing, and sending of messages, and is flexible enough to implement many new formats and applications.

2.3.2 HTML and the World Wide Web

As a content platform, the World Wide Web and the technologies that supply its content have been unparalleled in their usefulness and popularity. The heart of the Web is a set of pages, written in *hypertext markup language* (HTML) and rendered by and viewed with applications in various forms, known as Web browsers. These browser applications are clients that contact Web servers—which send pages to the browsers using *hypertext transfer protocol* (HTTP). These pages are connected to each other by hyperlinks on each page that, when followed by clicking the mouse or tapping a stylus on the link, cause the browser to automatically retrieve and display the page referred to by the hyperlink. HTTP uses the TCP/IP protocol suite.

Symbian OS supports Web technologies in two ways.

- It provides support for HTTP and HTML transport in its implementation of networking protocols. HTTP is implemented as a high-level protocol carried by several transport mechanisms. HTML is supported separately and assumed to be carried to a device by some transport means.

- It supplies applications that can be used as Web browsers. There are several Web browsing applications—from Symbian and third parties—that Symbian OS provides and supports.

2.3.3 WML and WAP

Browsing HTML content on the Web has been a great success for desktop and handheld computers. The intent of WAP is to move this success to mobile phone-based platforms. While a great idea, the browsing platform and its content need to be reworked completely for mobile, phone-oriented technology. First, microbrowsers have to be used to reduce the load on the smaller CPU and restricted memory. Then HTML needs to be reduced to fit into these microbrowsers, resulting in *wireless markup language* (WML).

As with technologies surrounding HTML and HTTP, Symbian OS supports WML and WAP in several ways.

- There is native library support for WAP transport and for WML.
- Symbian OS supplies applications for WAP browsing and rendering.
- It supplies tools and interfaces for building separate applications that implement WAP page retrieval.

2.4 Requirements of Communicators and Smartphones

Although there can be many kinds of communicators and smartphones, they have some common requirements that the designers of Symbian OS had to watch. It is difficult to design an efficient communication architecture that meets the needs of the future—especially when the future changes so rapidly. These communication platforms have dictated and directed this design. We will examine their needs, to see how those needs have influenced platform design.

To start with, it is important to mention what are *not* requirements. It is certainly *not* a requirement that these devices all look the same or even do the same things. These devices can come in all shapes and sizes, each implementing different views of computing and communication. Communicators and smartphones do not have to have the same hardware requirements; monochrome screens should be as acceptable as color and many different memory configurations are possible. They do not even need to have the same I/O devices

or configuration. In short, Symbian OS places no restriction on hardware design.

However, there are requirements outside of design constraints that Symbian OS devices place on a communications infrastructure. First and foremost, Symbian OS's communication architecture must be one that *supports many device designs* and is *extensible*. As previously mentioned, there is no restriction on size, shape or design. To accommodate many different designs, we do not want to have to completely redesign communications for a new communication method. We want a new method's implementation to fit in with existing implementations, and to be able to take advantage of all the same facilities as the other implementations.

As with most computing devices, these platforms require *robust* implementations. In fact, communicators and smartphones have special needs in this category. While larger computers can perhaps be reset easily or can tolerate errors without affecting other applications, these smaller devices cannot easily accommodate error conditions, nor can they be reset as easily. It does not work to "reboot" your mobile phone to correct a problem, or to restart a data stream or video transmission. These devices often use realtime communication services that charge by the byte or by the connection session. Therefore, restarting equipment or data transfers can cost the user an unacceptable amount of money.

Communicators and smartphones require an *efficient* implementation of communications. By "efficient", we mean both "fast" and "easy on resources". These factors are complicated by the wide variety of designs. No assumptions can be made about speed of processors, amount of memory, or specific communications available. We must have a modular, adaptable implementation, yet one that takes up a small space and runs fast.

Finally, one assumption we can make is that communicators and smartphones will run on batteries. Therefore, implementations must be easy on the resources. Battery power must be preserved, but access to communication ports will utilize power. Therefore, care must be taken to economize access while fully implementing standards.

So, we have a tall order. We must have a general, modular, adaptable infrastructure that supports the implementation of many different communication standards. This general implementation must be robust and efficient, while at the same time being extensible and easy on resources. This is the mandate for Symbian OS.

2.5 Implementation Techniques

As we begin to examine the architecture of Symbian OS, we will notice patterns in the way the architecture is implemented. Certain techniques are used in implementations; understanding these techniques will help to understand the communications architecture.

2.5.1 Client-Server Relationships

The way computers interact—especially in a TCP/IP network like the Internet—can be characterized as a *client-server* type of relationship. The client is the computer taking advantage of the service; the server is the computer providing that service. The client typically utilizes the service by sending requests to the server; the server provides the service typically by responding to these requests. The requests and responses take the form of messages sent back and forth between client and server.

A good example is the microbrowser in a WAP-enabled mobile phone. Figure 2.2 shows how this works. The phone calls up a WML page by sending an encoded request to a computer at the company that provides the service. This computer is called a *gateway server*, because it provides a gateway to the Internet for the mobile phone. This server, in turn, becomes a client by passing the request from the mobile phone on to the actual Internet server that houses the WAP page in question. This server happens to be providing an HTTP service, and answers the request by sending a Web page to the gateway server. The gateway server then translates the HTML page into WML, and encodes the response using the encryption scheme the mobile phone expects. Finally, the gateway server sends the response to the mobile phone, and the microbroswer displays the WML page on the phone's display.

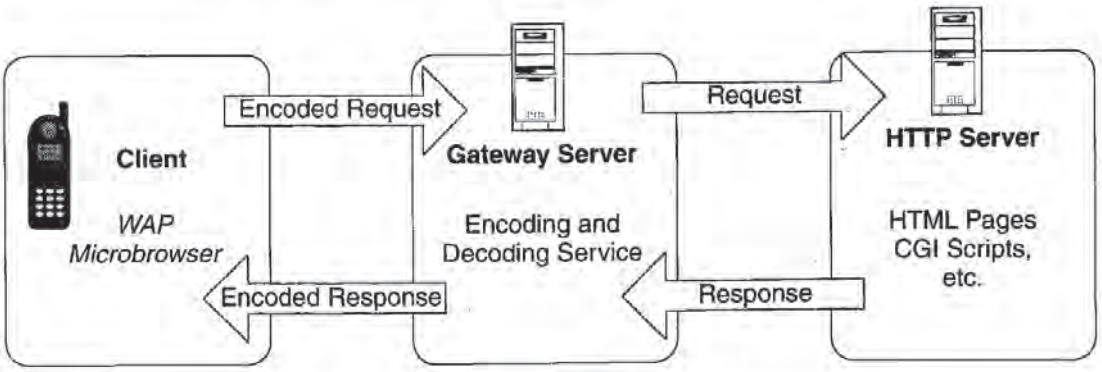

Figure 2.2 Client-server relationships in WML browsing.

Note that in the example, there were two clients and two servers. The gateway server was both a server to the mobile phone and a client to the HTTP server. This is an example of a *proxy server*, a server that represents its client on another network by becoming a client itself.

A variant of this relationship occurs in the Symbian OS kernel to protect and manage resources. In this case, the kernel acts as a server and applications act as clients. The middleware that we discussed in Chapter 1 is an example of this. Applications use a protected resource by connecting to the server that manages that resource. Once connected, applications or clients manipulate the resource through the resource's server (in an application's code, the resource is manipulated by using the server object's functions).

2.5.2 Asynchronous Events

Communications programming requires quite a bit of waiting. As a client, an application might wait for a response from a server. As a server, an application would always be waiting for client requests. An application would be likely to spend time waiting on devices—such as IR ports or Bluetooth chips—to transmit messages or to report back with data.

Because waiting is so frequent, a programming model has been developed for it: *asynchronous, event-driven programming*. This type of programming is asynchronous because responses do not arrive in fixed intervals (if they return at all). We use the idea of an "event" to represent the occurrence of something that an application might be waiting for in a communications exchange. In Symbian OS, there are many events that are specified to represent communications. We will discuss these in later chapters.

With asynchronous programming, there is an event defined as a timeout. A timeout is a good example of the care a programmer has to take with communications. Assuming, for example, that all communications will result in a response would be a mistake. Programmers must plan and design for the event that nothing happens when communications is attempted. We will discuss ways to build timeout exceptions in the way we approach communications.

This asynchronous event model is useful in several ways. It keeps applications from freezing while they wait for some event. It allows the system to interleave instructions more efficiently from concurrent processes/threads. It also allows the system to put the entire device to sleep in certain cases (e.g., all processes are waiting for events) to conserve battery power.

2.5.3 Active Objects

To accommodate the constant waiting imposed by the asynchronous nature of communications, Symbian OS uses the idea of *active objects*. An active object is a thread—a parallel object that executes alongside the main code—that makes communications requests and handles the event that is generated when the request gets a response. By building into the Symbian OS the concept of an active object as a thread, the designers of Symbian OS built an object that can handle waiting for responses from devices and not prevent other code from running.

An example of how an active object might be used is an email application. Email can be collected by sending a request to a POP3 (Post Office Protocol, version 3) server. A response is rarely immediate, however, especially if there are many messages to pick up. If an application were not to use active objects, the application would quite likely freeze up while waiting for all the messages to download. This situation would occur because the application would be concentrating on the data exchange with the POP3 server, and not on listening for stylus taps or updating the screen. However, if the application used an active object for communicating with the POP3 server, then it could respond to the user at the same time it was waiting for messages. Active objects enable more interaction with applications and a clean interface for handling situations that might arise during communication.

2.6 Summary

This chapter opens a section of this book on the basic concepts and design of Symbian OS's communication architecture. Specifically, we covered the following:

- What a "communications architecture" actually is
- What technology is supported by Symbian OS—from a *transport* and a *content* perspective
- What requirements communicators and smartphones make on Symbian OS's communications architecture
- What implementation (or programming) techniques are used by Symbian OS: client-server relationships, event-driven programming, and active objects

The next chapter will introduce the communications structure of Symbian OS from several different perspectives, taking a close look at the elements that form its foundation.

3

Several Introductions to the Architecture Infrastructure

In Chapter 2, we had an introduction to the communication architecture of Symbian OS. We stated in that chapter that this architecture "is more than simply infrastructure". While this is certainly true, the infrastructure of communication in Symbian OS is a foundational part of the architecture and we need to discuss it. This chapter takes a look at the concepts that comprise the infrastructure of Symbian OS's implementation of communication from several different perspectives.

The communication infrastructure weaves several common themes through the various interfaces and layers of its implementation. We look at these common themes in this chapter. We will begin with an overview of the infrastructure by looking at the components used to build it. Then we will take a close look at the client-server model in Symbian OS, which is at the center of the infrastructure. Next, we will overview the communication database, which is at the heart of accessing peripherals. We will conclude by examining two issues tied closely to the infrastructure: synchronization and security concerns.

3.1 Basic Infrastructure Building Blocks

A good way to start looking at the infrastructure is by examining its basic components. This will also serve as an introduction to the implementation.

Let's start by considering a very generic form of the communications infrastructure. Figure 3.1 contains a generic diagram of the infrastructure. Consider this diagram as a starting point for an organizational model. At the bottom of the stack is a physical device, connected in some way to the computer. This device could be a

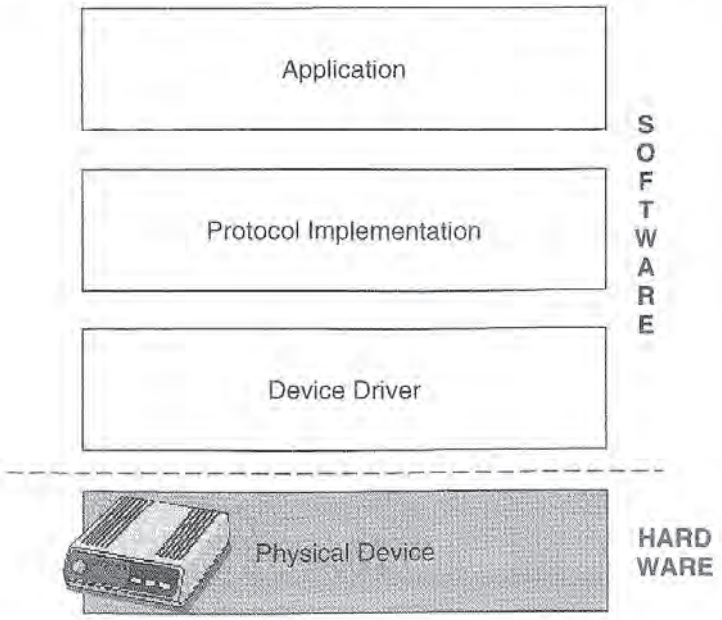

Figure 3.1 A generic API diagram.

mobile phone modem or a Bluetooth radio transmitter embedded in a communicator. Since we are not concerned with the details of hardware in this book, we will treat this physical device as a "black box", an abstract unit that responds to commands from software in the appropriate manner.

The next level up—actually the first level we are concerned with—is the *device driver* level. The software at this level is concerned with working directly with the hardware via a fixed, standardized interface to the upper software layers. The software at this level is hardware-specific, and every new piece of hardware requires a new software device driver to interface with it. Symbian OS comes with a set of device drivers for commonly used pieces of hardware (e.g., internal modems and printers). Different drivers are needed for different hardware units, but they all must implement the same interface to the upper layers. The protocol implementation layer will expect the same interface no matter what hardware unit is used.

Standards play a major role with device drivers. Hardware is becoming increasingly standardized, and sometimes one device driver can manage several pieces of hardware because they all abide by the same standard. For example, many modems can be controlled by a single device driver. In addition, protocol implementations are increasingly assuming device driver standards. Standards such as IrCOMM, for instance, are becoming widely supported and therefore must be incorporated in the device driver layer.

The next layer up is the protocol implementation layer. This layer contains implementations of the protocols supported by Symbian OS. These implementations assume a device driver interface with the layer beneath and supply a single, unified interface to the application layer above. This is the layer that implements the Bluetooth and TCP/IP protocol suites, for example, along with the other protocols discussed in Chapter 2.

Finally, the top layer is the application layer. This layer contains the application that must utilize the communication infrastructure. The application does not know much about *how* communications are implemented—however, it does do the work necessary to inform the operating system of which devices it will use. Once the drivers are in place, the application does not access them directly, but depends on the protocol implementation layer APIs to drive the real devices.

3.2 A Closer Look at the Infrastructure

Now let's take a closer look at the layers in Symbian OS communication infrastructure. Figure 3.2 contains a diagram based on

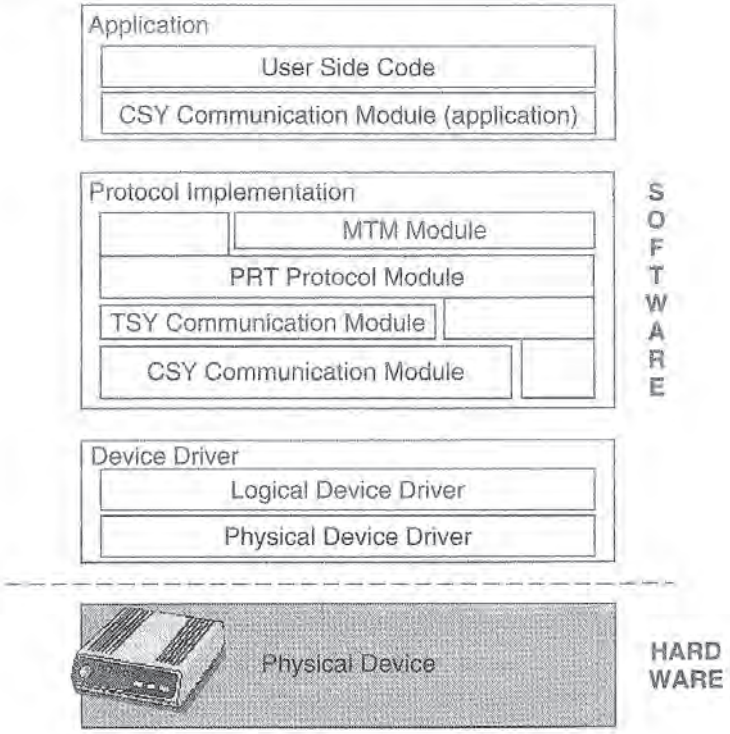

Figure 3.2 More detailed API diagram.

the generic model in Figure 3.1. The blocks from Figure 3.1 have been subdivided into operational units that depict those used by Symbian OS.

3.2.1 The Physical Device

First, notice that the device has not been changed. As we stated before, Symbian OS has no control over hardware. Therefore, it accommodates hardware through this layered API design, but does not specify how the hardware itself is designed and constructed. This is actually an advantage to Symbian OS and its developers. By viewing hardware as an abstract unit and designing communication around this abstraction, the designers of Symbian OS have ensured Symbian OS will handle the wide variety of devices that are available now and that it can accommodate the hardware of the future.

3.2.2 The Device Driver Layer

The device driver layer of Figure 3.1 has been divided into two layers in Figure 3.2. The *physical device driver* (PDD) layer interfaces directly with the physical device, through a specific hardware port. The *logical device driver* (LDD) layer interfaces with the protocol implementation layer and implements Symbian OS policies as they relate to the device. These policies include input and output buffering, interrupt mechanisms, and flow control. The division of these layers represents a division in implementation, where the PDD implementers can focus on an efficient and correct hardware interface, and the LDD implementers can work to perfect the interface with the upper layers to maximize performance.

As an example of this division of responsibilities, consider the serial interface. There are several serial device types that can be connected to an Symbian OS device. The IR port and the RS232 port are both serial devices and can use the same LDD—called `ECOMM.LDD` on Symbian OS version 6.1. These ports are serial ports and use the same policies with respect to issues such as flow control. However, their PDD modules will be different: one will service the RS232 port and one will service the IR port.

3.2.3 The Protocol Implementation Layer

Several sublayers have been added to the protocol implementation layer in Figure 3.2 from Figure 3.1. Four types of modules are used for protocol implementation; these are itemized below:

- *CSY Modules:* The lowest level in the protocol implementation layer is the *communication server* or CSY, module. A CSY module communicates directly with the hardware through the PDD portion of the device driver, implementing the various low-level aspects of protocols. For instance, a protocol may require raw data transfer to the hardware device or it may specify 7-bit or 8-bit buffer transfer. These "modes" would be handled by the CSY module.

  Note that CSY modules may use other CSY modules. For example, the IrDA CSY module that implements the IrCOMM interface to the IR PDD also uses the serial device driver, ECUART CSY module.
- *TSY Modules:* Telephony comprises a large part of the communications infrastructure, and special modules are used to implement it. *Telephony server*–or TSY—modules implement the telephony functionality. Basic TSYs may support standard telephony functions, e.g., making and terminating calls, on a wide range of hardware. More advanced TSYs may support advanced phone hardware, e.g., those supporting GSM functionality.
- *PRT Modules:* The central modules used for protocol implementation are *protocol modules* or PRT modules. PRT modules are used by servers to implement protocols. A server creates an instance of a PRT module when it attempts to use the protocol. The TCP/IP suite of protocols, for instance, is implemented by the `TCPIP.PRT` module. Bluetooth protocols are implemented by the `BT.PRT` module.
- *MTM Modules:* As Symbian OS has been designed specifically for messaging, its architects built a mechanism to handle messages of all types. These message handlers are called *message type modules* or MTMs. Message handling has many different aspects, and MTMs must implement each of these aspects. User Interface MTMs must implement the various ways users will view and manipulate messages, from how a user reads a message to how a user is notified of the progress of sending a message. Client-side MTMs handle addressing, creating and responding to messages. Server-side MTMs must implement server-oriented manipulation of messages, including folder manipulation and message-specific manipulation.

These modules build on each other in various ways, depending on the type of communication that is being used. Consider Figure 3.3, which depicts some example module relationships. Implementations of protocols using Bluetooth, for example, will use only PRT

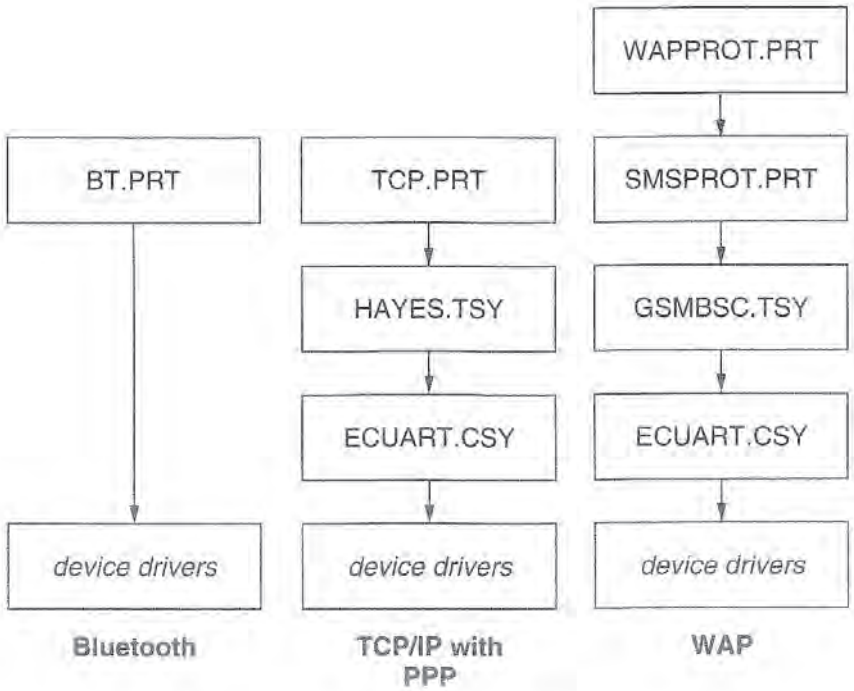

Figure 3.3 Example communication module relationships.

modules that communicate directly to device drivers. Certain IrDA protocols will do this as well. TCP/IP implementations that use PPP will use PRT modules and both a TSY and a CSY module. The WAP protocol stack uses a WAP PRT on top of an SMS PRT, which in turn is built on a GSM TSY and some kind of CSY (ECUART, IrCOMM, or RFCOMM).

Look for Yourself

See for yourself which modules are installed on your emulator or Symbian device. You need to look on the Z: drive, which Symbian OS uses to designate the ROM. To do this, use the Web browser in your emulator or Symbian device and activate its main menu. Choose `File`, then `Open`, then `Open Web Page` (on the Quartz emulator, you must choose `Web`, then `Open Web Page`). In the file dialog that opens, type `Z:` in the text field under the `Open URL Address` label (`Location`, for Quartz). When you press the Enter key or choose the Ok option, the Web browser will present you with a display that holds the contents of the ROM.

On the Quartz emulator, there is an easier way. The Symbian OS 6.1 Quartz emulator comes with a file browser interface. It is called "QFileMan". Tapping the icon brings up the interface.

You can browse the `Z:` drive by tapping the righthand menu and choosing the correct drive.

Use the interface to explore the ROM. In particular, look under `Z:\System\LIBS`. Look for the CSY, TSY, PRT, and MTM modules that are delivered with your emulator or device.

3.2.4 Infrastructure Modularity

We should take a moment here to appreciate the modularity of the communication infrastructure design. The "mix and match" quality of the layered design should be evident from the examples just given. Consider the TCP/IP stack implementation. A PPP connection can go directly to a CSY module or choose a GSM or regular modem TSY implementation, which in turn goes through a CSY module. When the future brings a new telephony technology, this existing structure still works, and we only need to add a TSY module for the new telephony implementation. In addition, fine tuning the TCP/IP protocol stack does not require altering any of the modules it depends on; we simply tune up the TCP/IP PRT module and leave the rest alone. This extensive modularity means new code plugs into the infrastructure easily, old code is easily discarded, and existing code can be modified without shaking the whole system or requiring extensive reinstalls.

3.2.5 Application CSYs

Finally, Figure 3.2 has added sublayers to the application layer of Figure 3.1. There are CSY modules that applications use to interface with protocol modules in the protocol implementations. While we can consider these as parts of protocol implementations, it is a bit cleaner to think of them as assisting applications. An example here might be an application that uses IR to send SMS messages through a mobile phone. This application would use an IRCOMM CSY module on the application side that uses an SMS implementation wrapped in a protocol implementation layer. Again, the modularity of this entire process is a big advantage for applications that need to focus on what they do best and not on the communications process.

3.3 Client-Server Relationships

Another way to examine the communication infrastructure is by looking at the client-server relationships built into it. This section will examine the servers; the next section will look at the clients.

3.3.1 Kernel Servers

In general, Symbian OS uses kernel servers to protect and manage shared resources and services. Servers form a gateway that processes must go through to access communication functionality. If the operating system was only able to run one program at a time, run that program until it was done, and move on to the next one, servers would not be necessary because only one access at a time would be possible. But Symbian OS is a multitasking operating system, allowing multiple processes to run at the same time. And Symbian OS is also multithreaded, allowing multiple threads of control to run in parallel within a single process. All this concurrency could wreak havoc on communication resources. Imagine three Web browsers, each using the TCP/IP stacks and each fighting for the communication ports at the same time.

The use of servers, then, cuts down on the amount of memory and CPU resources needed by supplying access to resources that applications do not have to implement. Using servers also correctly channels the accesses from multiple processes and threads into proper usage of resources.

In Symbian OS, servers are accessed by clients in the following way:

- Servers must first be started if they are to serve their gateway function. Only a few servers are started when the operating system initially boots, to prevent slowing down a system with rarely used servers. The other servers must be started explicitly before an application may use them. To help application programmers, Symbian OS allows the starting of servers that are already started (only one copy will be left running); this means applications can blindly start servers without checking if the one they need is running.
- To use a server, clients must establish a connection to the server they need. Within the communications infrastructure, this client-server connection is facilitated by a *socket*. Servers listen for socket connections, and have the option to accept or deny connections. Most of the time, servers will accept socket connections. Decisions to deny connections can be based on information that comes with the connection request (e.g., the identification of the sending computer) or on local information (e.g., the number of connections already open).
- Once a socket is open, the server waits for requests to arrive over that socket. Requests are simple bytes of data, interpreted by the server in special, designated ways. The server then services

those requests and sends responses back to the client, again in the form of bytes of data written to the socket.

- When the exchange is done, the socket is closed. Either side can close the socket, but Symbian OS notifies both sides of the closure no matter which side initiates it.
- A server might stay active for a time to service other requests or might terminate immediately to preserve resources.

There are several servers used throughout the communication infrastructure. These servers include:

- *Serial communications server:* The serial communications server provides clients with a consistent interface that manipulates serial devices. The serial communications server manages the built-in serial ports and the IR port, using the ECUART and IrCOMM communication server modules to interface with lower layers.
- *Telephony server:* The telephony server manages services for initiating and terminating voice, fax, and data calls. It provides an API for accessing telephony services implemented through either GSM mobile phones or landline modems (using Hayes standard AT modem commands).
- *Socket server:* The socket server manages socket connections and provides a programming interface for a variety of protocols to clients. It also allows new protocols to be inserted into the socket framework. Standard interfaces are provided for Internet and IR protocols, as well as for the Psion Link Protocol (PLP) which is used for PC synchronization.
- *Communications database server:* As the central repository for information regarding configuration settings for devices, the communications database server serves an important function. It guards the source of information for devices and manages access to this information set. This information set includes data pertaining to phones, modems, protocols, and Internet service provision information (network access phone numbers, Internet addresses and physical locations). In addition, details of device drivers available to the system servers and the Symbian OS kernel are available through this server. This server is an instance of the Symbian OS DBMS server (see Section 3.4 below).
- *Other servers:* There are other communications servers that Symbian OS uses that serve a less common, but important, role. These servers include the fax server, which is used to send and receive faxes, and the resolver server, a service that resolves

names of Internet sites to IP numbers. The concept behind these is the same as with all servers; one must connect to the server, then use the connection object's API to access the information being protected.

You might have noticed that each of the descriptions above use a phrase something like "...this server provides an API for accessing...". This might seem strange, as we have talked about "connecting" to servers rather than using a server's API. It is, however, the duty of a server to provide a standard interface for the usage of the services it manages. Most times, even though a client sets up a socket to the server, the real interaction with the server takes place through function calls to a server object, not by explicitly constructing requests for the server. This will come through when we discuss the details of programming.

You might also have noticed that there seem to be a few things missing from the above list. For example, Bluetooth, which is featured quite prominently on Symbian OS, is not implemented via its own server. It is generally the case that the above servers are all that Symbian OS needs. If the technology you are looking for is not in the above list, it is likely implemented by other servers. Bluetooth, for example, is implemented using its own serial communication module (look for `btcomm.csy`) but using the serial communications server.

3.3.2 Clients

Clients are the users of the resources that servers are built to protect. They contend with each other to use resources. Any process or thread can be a client of a server; in fact, because threads can be clients, a process might even contend with itself for a resource. It is the server's role to manage this contention for resource access.

To become a client, a process or thread needs to go through the following steps:

- Clients must first make sure the drivers—both the LDD and the PDD parts—are loaded by Symbian OS kernel. Different drivers will be loaded, depending on how a client wants to access a particular resource.
- Next, a client needs to start a server in order to communicate with it. Remember that the number of running servers is typically carefully planned, and that a client's server may not be running when it is needed.

- A client must then connect to a server. This is through a call in the socket object's interface (typically, a call to `Connect()`). A socket is opened to the desired server and data may flow between the client and the server.
- Once the connection is made, interaction occurs in a request-response cycle. The client makes requests to the server by

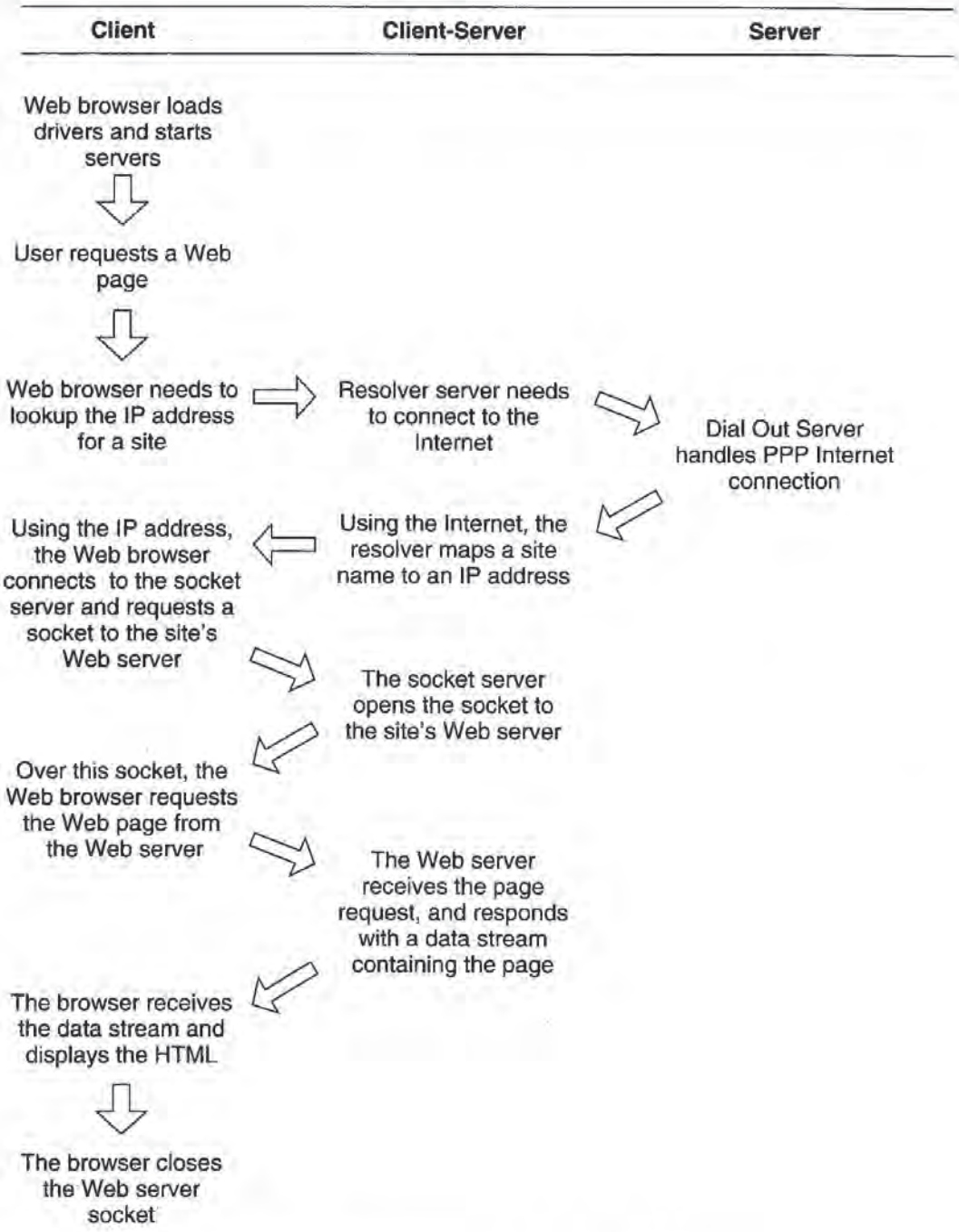

Figure 3.4 A client-server interaction for a Web browser.

sending it messages (in the form of data); the server reads the request and sends the responses back.

- Eventually, the client will complete its use of the server and will close the connection.

An obvious example of a client is a Web browser application. Figure 3.4 depicts a typical sequence of client-server interactions that constitute reading a Web page. The sequence we outlined above is followed. Other examples in the Symbian OS standard application set include email clients, printing applications, and Bluetooth data exchanges.

There are also other, hidden clients that use servers hidden in the internals of the communications infrastructure. For example, consider Figure 3.4 again. The resolver server was the first to initiate a network connection, thereby becoming a client itself when it turned to the communications database and the socket server to establish connections with the Internet.

3.4 The Communications Database

A third way to view the communication infrastructure is through the communications database. The communications database is a standard Symbian OS database, manipulated by the Symbian OS DBMS system. It stores all information that should need to be referenced about components of the communication infrastructure. This information includes phones, modems, protocols, network access phone numbers, Internet addresses and physical locations. The database is also used to record the settings current at any time.

From a user's perspective, this database is maintained through the Control Panel in Symbian OS user interface. From an application perspective, it is manipulated via the conventional Symbian OS API for relational databases. The communication infrastructure does provide a streamlined way to access the database, but this API amounts to convenience methods that implement operations that could be done using standard means. We will cover this in Chapter 18.

The communications database is used by applications that require component configuration information. By using the communications database, an application shares the user's intentions for the device with other applications and at the same time is truly device independent. For example, a program that needs access to a serial service can reference the database to identify the default module to load, using a dialog box to allow the user to confirm the setting.

Browse the Database

A good way to become familiar with the common components of Symbian OS is to examine this database. While browsing the actual database itself entry by entry is probably not too informative, the Quartz emulator includes a tool that allows you to browse and edit the database by categories of use.

To start this tool, you need to execute the command `setupcomms` in the DOS command shell. This will bring up a window like the one shown in Figure 3.5. This window will allow the browsing of categories that are used in the communications database. Beware, however, that changing anything using this interface will have an effect on the way applications work in the emulator.

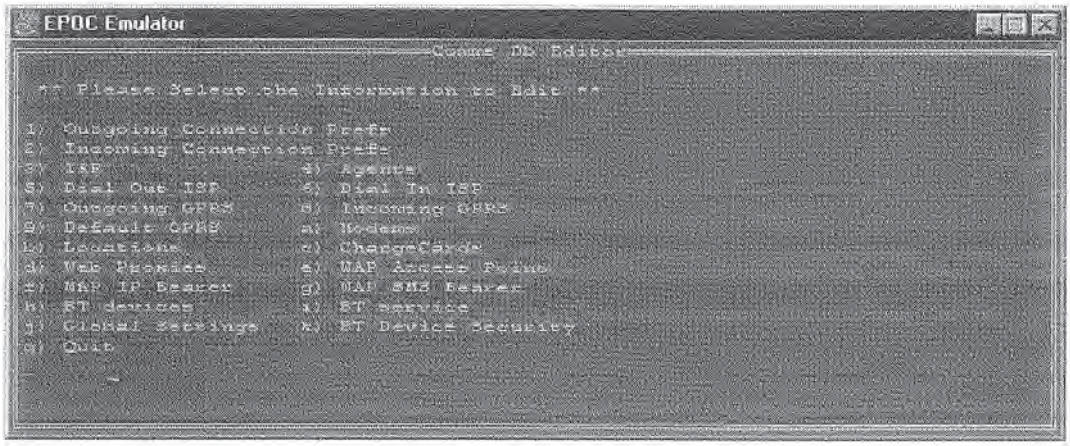

Figure 3.5 Browsing the communications database using `setupcomms`.

3.5 Infrastructure Issues

As we look through the infrastructure, a few global issues crop up that should be examined. Other issues will come up and we will deal with them later; synchronization and security matters are so woven through the infrastructure that we should deal with them now.

3.5.1 Synchronization

Many Symbian devices are commonly used in connection with other computers—whether desktops, laptops, or other handheld computers. It is typical in these situations for the two computers to share information. The Symbian platform was designed from the ground

up with this type of cooperation in mind. For example, sharing agenda information between a handheld Symbian device and a larger desktop computer running Microsoft Outlook is an everyday event for many people. This synchronizing of information is such a common event that it is accommodated in the communication infrastructure.

In a different environment, we might share information by simply accessing the same files over a network. Two applications on two different machines might share one file and each manipulate that file. In this case, a server would probably be involved, with the two machines wanting to share information acting as clients to a third machine acting as the server.

The situation we are focusing on, however, is fundamentally different. We do indeed have two computers, but these computers are disconnected from each other most of the time. They do not need to *share* a file; they need to *synchronize information*. File sharing would imply that the two computers are running the same application and therefore need the same file. In our case, we cannot assume that the two computers are running the same application; in fact, we assume that they are not. In this case, we need to synchronize information, i.e., we need to make sure that both sides have equivalent information as specified by their specific applications. To do this, we need to (a) transfer files between the two computers, (b) determine if files are up-to-date, and (c) convert files from one format to another. Therefore, synchronization implies file transfer and file conversion.

In Symbian OS, because many of the actions involved in synchronization are also involved with other integral activities, synchronization was implemented using a new protocol. The Psion Link Protocol (PLP) was implemented to transfer files and to enable remote control of a Symbian device. PLP works over serial media—the serial port or IR port—and requires an application on the other computer that implements PLP as well. Psion, for example, provides a program called "PsiWin" to provide synchronization and other browsing tools to its devices. Symbian provides a set of tools to implement this remote application.

To incorporate this new protocol into Symbian OS communications infrastructure, we simply need a PLP protocol module. Since PLP uses serial communication, no new device drivers are needed. PLP is implemented by the `PLP.PRT` protocol module.

3.5.2 Concerns over Security

By definition, communications move data between two computing devices or phones. Any time this transfer happens, there is

opportunity for that data transfer to be observed or even disrupted by a third party. We can control the computers themselves and can even watch them at all times, but once the data leaves the computer en route for a new one, control is lost. We can make very few assumptions about the data as it makes its way to its destination: Is it being observed? Is it being changed in some way?

Consider how insecure various media are. Serial cables are by nature point-to-point connectors, and it is difficult to discern the data transferred by somehow "listening" to the outside of a cable. The nature of serial communication is for the devices to be relatively close to each other, so any tampering with the connection media would be noticed. Infrared communication, however, allows snooping of at least one side of the communication without big changes to the environment. A machine could certainly be situated within the infrared transmitter range and pick up one side of the conversation. Networks are inherently insecure. Because of the passive, multipoint nature of networks, all network data is accessible to all network nodes. Low level network protocols, like Ethernet, are based on the fact that a node will ignore data unless that data is targeted at the node. But all Ethernet cards have a "promiscuous" mode, where all data can be read and analyzed. Wireless data transfer and data networks are extremely insecure. While a machine must be physically connected to a wired network in order to snoop it, the requirements for snooping wireless data are simply that a machine has a receiver that can read the wireless data. Such data—from mobile telephone traffic to network data to Bluetooth conversations—can be read and analyzed from a hidden location or using small reader equipment.

So now that we are convinced that our data transmissions can be intercepted, read, and changed, how do we protect our data? It is important to first realize what we *cannot* do. For most data transmissions, we cannot control the transmission's course or the eyes that view the transmission. It is simply not feasible to know how the data is transmitted every step of its journey. Data over the Internet, for instance, will move through many different pieces of routing equipment before they will reach their destination. Each piece of routing equipment might be in a secure location, or might be out in a public area. The sender cannot control where or how the data gets transmitted.

What we do have control over is the data itself. Most methods to ensure the secure transmission of data utilize *encryption*, a method of data encoding that renders the data unreadable for those readers that do not know how to decode it.

Encryption can occur at different levels in the protocol stack. Encryption can be implemented, for example, by applications themselves. The data is encoded before transmission by the sender and decoded after it reaches its destination. An example of this is the Secure Hypertext Transfer Protocol (SHTTP), a protocol used by Web servers and browsers to pass important data (like credit card information). A Web server that uses SHTTP will encode the data, then send it to the Web browser it is connected to. The Web browser will receive the information, then decode it and display it. Data that goes to the Web server follows the reverse route.

Another kind of implementation encrypts the data in lower protocol layers after it is transmitted. In this approach, the application really does not know if the transmitted data has been encrypted or not. Lower layers encrypt and decrypt the data. An example of this is the stack of protocols that make up WAP. In the middle of the protocol stack is the WAP Transfer Layer Security (WTLS) layer. This protocol layer encodes any data it receives from the upper layers, passing that encrypted data on to lower layers, and decodes any data it receives from lower layers, passing that data on to upper layers. The advantage to inserting a new layer in the protocol stack to do this encryption is that *any* application can use encrypted data, not just those that have encryption built in.

While encrypting data is the most secure method of protecting that data, it is not the only method used. Most other methods rely on a form of *identification* of the user through *authentication* methods. Identification involves a way for the user to identify him- or herself, for example, by having the user type in a user name. Authentication involves verification of an identity by some unique information included with the username—say, a password—that can be stored and known only by the user and the authentication source. This type of security method is woven into protocols—PPP uses forms of authentication in its protocols, for instance—and is typically used to guard the entrance into time-shared computer systems. (Many Symbian devices use password protection upon power-up to prevent unauthorized access to the device.)

Using authentication as a security measure has been problematic. With the powerful computers that we have today, passwords can be easily attacked and discovered. Systems can be compromised by attempting to gain access repeatedly with different passwords. Many "canned" programs exist on the Internet for just this kind of entry. Some systems compensate by not using passwords and relying on other forms of unique information, such as fingerprints or retinal scans.

Therefore, most communications protocols and even system access programs are turning to data encryption for their security. Encryption will feature in many parts of the communication infrastructure, and we will point it out through this book as we encounter its use.

3.6 Summary

This chapter serves as an introduction to the infrastructure of the Symbian communication architecture. In this chapter, we looked at the infrastructure from four different angles:

- We looked at the building blocks—the basic components—of the infrastructure. In successively closer looks, we overviewed what the components are, how they fit together and what they do.
- We looked at the server side of the client-server relationships built into the infrastructure. We looked at the client side of the architecture's client-server relationships.
- We finished by overviewing the communications database used by Symbian OS and by applications to store information about the communications system.

We finished the chapter by introducing two issues that are important to the communication's architecture: data synchronization and system security.

Now that we have introduced the communications architecture, the next two chapters will examine the technologies that the architecture supports. This is broken down into *transport* technologies, covered by Chapter 4, and *content* technologies, covered by Chapter 5.

4

Transport Technologies

I do not particularly like to use the word "technologies" to describe the applications built on top of the communication infrastructure. It has become a buzzword, and has almost graduated from useful, everyday speech to marketing talk. However, Webster's dictionary defines this word as

> *the practical application of knowledge especially in a particular area*

or, in another way,

> *a manner of accomplishing a task especially using technical processes, methods, or knowledge*

Both of these definitions make sense. In this chapter, we will take the second definition: we will use technical methods to accomplish moving content data between machines. So I will use the term, but I will make every attempt to be descriptive, and steer clear of buzzwords.

In the last chapter, we looked at the communication infrastructure of Symbian OS from several different perspectives. The infrastructure is very important; it provides the foundation for the support of communication technologies. These technologies, of course, are what most people end up using and are the reason a Symbian OS device exists in the first place. So now that we have established the infrastructure, let's take a look at the technologies that make up the communications architecture.

As I have stated before, communication technologies fall into two categories: transport and content technologies. These are the technologies of moving data between machines and those that provide the useful information and applications on a Symbian

device. We deal with transport technologies in this chapter and content technologies in the next.

Different technologies exist because there are several different media used to transmit information. We will start this chapter by looking at the various media in use by Symbian OS (this will be brief; we have already seen this information in previous chapters). Then we will spend time on the details of communication transport methods. Along the way, I hope to dispel some myths about each technology and set expectations about its capabilities.

4.1 A Review of Communication Media

Communication media—the various hardware conduits that convey information—influence the software methods that are used to convey information. Transport technologies evolve in ways that both exploit and circumvent the hardware. So it will serve us well to review the communication media and look at their uses as well as their restrictions. Communications occur using both wired and wireless methods.

4.1.1 Wired Media

Wired media—cables, to be specific—have a long history by computer standards. The first cabled communication using computers goes back to the first computers. While the ENIAC did not have components, the UNIVAC—the first commercial computer, produced in 1951–contained components that communicated with each other using cables. Wired media take several forms: serial, parallel, telephony, and network media.

The concept of serial communication is simple. The serial port sends and receives information one bit at a time. Although this may seem slower than sending bytes all at once, it is simpler, less error-prone, and can be used over longer distances. Serial cabling, for example, can extend as much as 1200 meters, while parallel communication should only go for 20 meters. Serial communication is asynchronous, and data can be sent and received over the same cable at the same time (this is full-duplex communication, as we defined in Chapter 2).

In its simplest form, a serial communication cable needs only three wires. One wire receives data; one wire sends data; and one wire is an electrical ground. The two signal wires are crossed so that the sending wire becomes the receiving wire at the other end and vice versa. This kind of cable is called a "null modem"

cable, from the fact that this kind of cable is meant to emulate the signaling of a line without a modem. The serial cable used to connect a modem to a serial port, on the other hand, is wired "straight through"—meaning that the wires are not crossed as the wires in a null modem cable would be.

While only three wires are necessary, a few others are typically included. Two wires for synchronizing the exchange of data (hardware "flow control") are typically used, and four more for indicating readiness to send or receive data (see Section 4.2.1 below). The result is a set of nine wires. These wires are packaged in several different ways; a 9-pin configuration was popularized because it is used on PCs; a 25-pin configuration is also very widely used because it was the first generalized standard (although it does contain unused pins). Many different configurations are possible, because the standards that govern serial communications specify what types of signals should be used, not a specific pin configuration.

The Psion Data Port

The serial cable that Psion chose to use on many of their devices (e.g., the Series 5mx and the netBook) has a 9-pin connector on the PC side and a unique configuration on the Psion device side. This type of connector is called a *fast serial* "Honda" connector—Psion claims that it complies with the RS232 standard. While implementing serial communications, it has a different pin configuration than the 9 or 25 pin specifications for serial cables.

To find out more about this connector, check out information from the Web and the references in the bibliography. As you discover more, it might be helpful to draw a diagram of the Psion data cable, using a null modem cable as a base point.

The medium for parallel communication moves data bytes at a time. Because of this, there are many more wires in the cable. Parallel communication requires eight data wires, eight grounds for the eight data wires, and several more wires, including power for the data transfer. The result is between 25 wires (parallel printer cable) and 36 wires (Centronics parallel cable). Because of this complexity and the power requirements, parallel communication is not typically implemented on handheld devices.

Telephony requires only two wires: a sender and a receiver. These wires are housed inside a standardized connector, which has room for several wires and allows multiple telephone lines to be

included in the same connector for multi-line telephones. For our purposes, modems use only one telephone line, and therefore only two copper wires.

Wired network media take several forms. While past networks have used many different media types, currently there are three forms that are used. Two older forms are called "thicknet" and "thinnet". Thicknet is a 50 ohm coaxial cable, 0.4 inches in diameter. Thinnet is 0.2 inches in diameter, with the same impedance measure. Both cables are coaxial, a type of cable that includes one wire that carries a signal surrounded (after a layer of insulation) by metal braiding that carries another signal, effectively forming two channels. The outer channel also serves as a ground. Both of these forms of network cabling have fallen from grace, since they are inflexible and difficult to use in many situations. Thicknet is still used in enterprise network backbones.

The most popular wired medium for networks is called "10baseT" or "twisted pair" cables. Twisted pair is comprised of two ordinary insulated copper wires, twisted around each other to eliminate crosstalk interference. The cabling can be made of standard wiring (called "category 5"), and is normally terminated with standardized jacks. This makes connecting and disconnecting cables very easy. With this type of cabling, network hubs are needed. Cables connect the computing device to the hub, and the hub serves as the conduit for network data to move between cables or on to a larger network.

Thicknet, thinnet, and 10baseT cabling can handle network data that transfers at conventional speeds up to 100 Mbits/second. The newest technology has achieved Ethernet speeds of 1 Gbps.

4.1.2 Wireless Media

There are two alternatives we can choose from if we want to communicate without wires. The first is to use infrared; the second is to communicate by radio.

Infrared technology uses light with wavelengths longer than those of visible light, but shorter than those of radio waves. To compare it to other wireless technology, we would look at the frequency of this medium, measured in terms of terahertz, or trillions of cycles per second. They are just above radio-based wireless, which operates in the gigahertz (GHz) range.

Infrared devices can operate between two devices within a line of sight over a maximum distance of 5 meters. This has more to do with the transport technology than the medium, however. We will say more on this later.

All wireless technology other than infrared sends its data via radio. The main differences between the radio-based wireless technologies are the radio frequency they use and how they use that radio frequency. They operate at either 2.4 or 5.0 GHz. As the frequency increases, the bandwidth capability increases, but the range decreases. These are unlicensed bands, and many different technologies operate in them. For instance, cordless telephones, wireless networks, and Bluetooth devices all operate in the 2.4 GHz range and can easily interfere with each other.

4.2 Serial Technologies

4.2.1 Serial Standards

Several standards have been developed which use serial cables for serial communications. By far, the standard that has proved most versatile and durable is the RS-232 standard. Developed in 1960 by the Electronics Industry Association, the RS-232 standard was adopted by PC manufacturers and, because of this, will be around for a very long time. Symbian OS has always supported the RS-232 standard.

Sending an 8-bit word in a serial sequence requires both sides of the communication to agree on a way to figure out where the word starts and ends. One solution is to use *synchronous* communication, where bits are sent on specific clock intervals, with a brief interval between each word. Once both sides agree on what the word size is, synchronous communication makes it easy to determine where the word is in the data stream. Missing or extra bits can be detected; the word separator can be used to determine this. Synchronous communication has problems, however, in a generalized environment where different computers from different manufacturers are used. Timing can easily fall out of sync, and if either end loses a clock signal for any reason, the communication stops dead.

Asynchronous communication is much more dependable in general situations. In asynchronous mode, we do not use the clock to determine where the word markers are or where the bits are. We need to mark the ends of a word sequence, and allow each side of the communication to synchronize on these marks. The only requirement is that both sides use the same bit transmission speed. This speed is called the *baud rate,* and is measured in bits per second.

We can now construct a *data frame* out of markers for the beginning and ending of a word and the word itself. We surround the data bits by special bits called the *start* and *stop bits*.

In asynchronous communication, errors are harder to detect and correct. We need a way to encode the data so that we can detect an error, because we cannot depend on timing. Serial communications uses a *parity bit* added to the frame to signal the presence of errors. This bit is set to make the number of 1s in the frame even or odd, thus making bit flips somewhat easier to detect (although even numbers of bit flips would go undetected).

As an example, consider sending the letter "A" between two computers using even parity with one start bit. The ASCII encoding for an "A" is 7 bits: 1000001. The number of 1s is even, therefore the parity bit must be set to "0". Using a 1 as a single start bit gives us the data frame in Figure 4.1. Note the empty bit position. This "space" is allowed since the next data frame synchronizes when a 1 bit is sent.

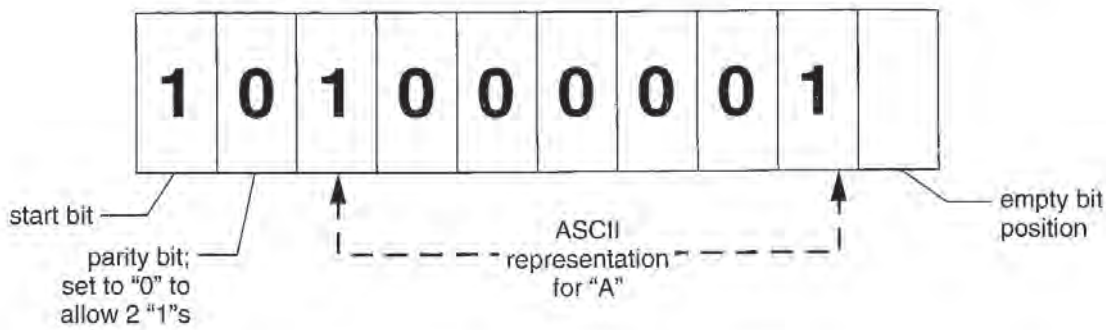

Figure 4.1 Serial dataframe using ASCII.

The RS-232 standard specifies a specific signal sequence for data exchange. The two sides of the transmission are called the Data Terminal Equipment (DTE) side and the Data Carrier Equipment (DCE) side. DTEs are typically originators of the communication, e.g., PCs or terminals. DCEs are typically communication devices like modems. A typical data exchange sequence might go like this:

- To indicate that it is initialized and ready, the DTE asserts the Data Terminal Ready (DTR) signal. To indicate a similar state, the DCE asserts the Data Set Ready (DSR) signal.
- When the DCE has a good carrier signal (e.g., a telephone connection to another modem), it asserts the Data Carrier Detected (DCD) signal.
- When a station wants to send data, it asserts the Request To Send (RTS) signal. When the receiver is ready, it asserts the Clear To Send (CTS) signal.

- The sender sends data using the Transmit Data (TD) signal; the receiver will receive over the Receive Data (RD) signal.
- So the sender will only send data when the DSR, DCD, and CTS signals are set. Likewise, dropping any of these signals will stop the data transmission.

It is important to note here that one side of the data transmission might be able to process data at a faster rate than the other side, even though the communication channel can deliver the higher baud rate. Therefore, we need a way to control the transmission, to tell one side to slow down or to stop while the other side catches up. This is done in one of two ways. One way is to use software, by sending special characters that are signals to stop and restart the data flow. In the ASCII character sequence, these characters are called XOFF (ASCII character number 19, a Ctrl-S) and XON (ASCII character number 17, a Ctrl-Q). While this is effective most of the time, it can be slow, and the characters can be lost or ignored. A more effective way is to use hardware flow control by manipulating the RTS and CTS signals. The signal on these lines must be asserted to send data; dropping the signal will stop the data flow. This is an effective and speedy method to control the data flow. This exchange of control signals is known as *handshaking*.

Debugging Hardware Signals

While software has its SDKs and debugging environments, there is very little that you can use if you are interested in debugging or examining hardware signals over a serial line. The most readily available tool is a *breakout box*.

Breakout boxes are small boxes about twice the size of a serial connector. The simplest models have a row of lights that turn on or off to indicate the state of one of the pins in the serial connector. Other models offer the capability to rewire the serial connection from inside the breakout box. This allows you to try various wiring configurations.

If you are interested in examining or experimenting further with serial signals, breakout boxes provide an easy method to do this. These are available at most places you would purchase serial cabling.

4.2.2 IrDA

RS-232 serial communications are relatively straightforward to understand. The data frame is simple and there is one function

for serial connection (i.e., to pass bytes to a receiver). Most other transport mechanisms are quite a bit more complicated. As a way to manage the complexity of the technology, these other mechanisms are designed in layers. Each layer focuses on a specific aspect of transport functionality.

IrDA technology is a good example of this effort to hide complexity. Although there is a physical device that communicates with other infrared devices via light, there is a protocol stack in Symbian OS that isolates this layer from other functionality. This stack is shown in Figure 4.2. Let's briefly consider each protocol layer.

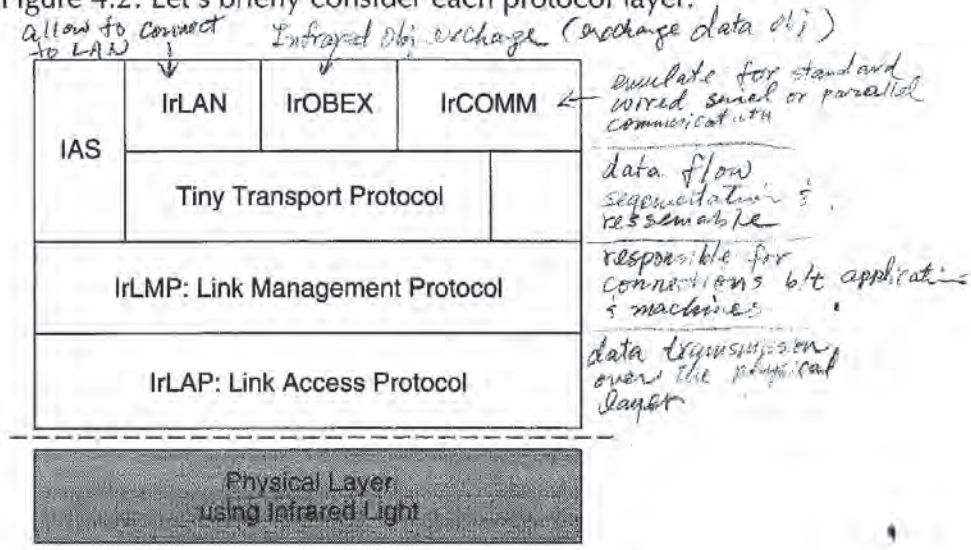

Figure 4.2 IrDA protocol stack.

The physical layer

The physical layer starts with an optical transceiver and deals with aspects of manipulating optical signals. Because the transmission medium is through optical signals, information is relayed in a serial manner, one bit at a time. And as with RS-232 serial communication, we must be concerned here about framing issues: start and end frame flags and encoding of data bits. The IrDA framing techniques are based on those used in networking standards, and therefore are more complicated than those used with RS-232 serial communication. A seven-bit sequence is used as the start and stop sequence, and a value is included with each frame that is computed from the bits in the frame and allows multi-bit errors to be caught.

As with RS-232 serial communication, IrDA communication can be either synchronous or asynchronous. Unlike RS-232 serial communication, either mode is used for different situations. Synchronous communication is faster, but is more prone to errors; asynchronous

communication is slower, but more flexible and adaptable to a variety of infrared equipment.

The physical layer is primarily a hardware layer, but a portion of it is usually implemented in software. To isolate issues of framing and protocol from physical issues of infrared light, a software layer is typically used here called the *framer*. The primary responsibility of the framer is to deal with framing issues, encoding and decoding outgoing and incoming frames and interfacing with layers above the physical layer. The framer also deals with hardware interface speeds and follows instructions from the IrLAP layer. The framer is included in the physical layer because the interface to the actual hardware is not standard, so changing speeds or reading and writing through the infrared device is still specified by the infrared manufacturer.

Using a Symbian OS Device as a Remote Control

A common question Symbian OS device users have is about the relationship between its infrared functionality and the infrared capability included on a remote control device. Both devices have infrared ports. Using your Symbian OS phone as a programmable remote control seems rather appealing. Unfortunately, the IrDA standards specify a different modulation frequency for IrDA compliant devices than the frequency remote controls use. So a program that implements a remote control device must bypass most APIs and reprogram the infrared driver outside of IrDA standards. This is a daunting task for all but the hardiest of programmers.

IrLAP: The Link Access Layer

The first all-software layer in the IrDA stack is the IrLAP layer, the Link Access Protocol Layer. The duty of the IrLAP layer is to provide reliable data transmission over the physical layer. By dealing with reliability at this layer, the upper levels in the stack are freed from dealing with it. We define *reliable data delivery* as delivery that is guaranteed either to happen or to fail with notice. No data will go undelivered without an error being generated and upper layers notified.

To ensure reliability, the IrLAP layer uses three mechanisms. First, it employs error detection by generating and monitoring the cyclical redundancy check (CRC) checksum. This value is computed, using an algorithm called "cyclical redundancy", based on the current string of bits that make up the data frame. If the

CRC checksum computed for the transmitted data frame does not match the checksum that accompanies the data frame, then an error occurred in transit.

The second mechanism used by IrLAP to ensure reliability is to establish a primary/secondary relationship between the sender and the receiver of an infrared transmission. The primary device initiates the connection/transfer, is responsible for the organization and control of the data flow, and deals with the errors that occur. The secondary device is only responsible for sending acknowledgement frames for data received. The primary in an IrLAP relationship obviously has much more work and management to do. Because of the acknowledgement-based protocol, the exchange occurs by turns, with each side taking a turn in the exchange of data. Neither side can talk for more than 500 milliseconds at a time before allowing the other side to take a turn (even if there is nothing to say).

The third reliability mechanism used by IrLAP establishes two modes of operation, based on the establishment of a connection between two machines. One mode, called Normal Disconnect Mode (NDM), is the default state of devices when they are disconnected. A device in NDM must check whether other transmissions are occurring before transmitting. If no IR activity is detected for more than 500 milliseconds, a device is free to try to establish a connection. Communication from the NDM state always follows the same serial configuration parameters: asynchronous communication at 9600 baud, using 8 bit frames with no parity bit set. By establishing this rule, each initial connection is made with the same parameters. Once an initial connection is established, the two sides often exchange different capability information, and they alter their configuration to the best one for both sides.

The other mode IrLAP establishes is Normal Response Mode (NRM). This is the normal mode of operation for connected devices. Once both devices are communicating and have exchanged configuration information, this mode allows the upper layers to assume the connection exists.

The IrLAP layer uses different framing techniques based on the speed of the IR connection. Three different framing methods are used for three different situations:

- Asynchronous connections can run from 9600 bps to 115 Kbps.
- Synchronous connections that use a network style of framing (synchronous data link control or SDLC) can run from 576 Kbps to 1.152 Mbps.

- Synchronous connections using a "retooled" network framing technique (known as pulse position modulation or 4PPM) can run up to 4 Mbps.

One final note should be made here about the functionality of IrLAP. It is possible to share the IR port between several different applications, each sending their own data to companion applications on a remote device. The IrLAP layer enables this by establishing an address for each data stream. Because IR connections are point-to-point, the connected devices do not really need addresses. However, by tagging data streams with addresses, a single IrLAP connection can be multiplexed between upper layer applications.

IrLMP: The Link Management Layer and IAS

The Link Management Layer in the IrDA stack assumes the reliable connection established at the IrLAP layer, and is responsible for the connections between applications on different machines. By multiplexing the IrLAP link, implementing device discovery, and managing connections between applications, the IrLMP layer connects applications to one another.

The connections at the IrLMP layer are made between Logical Service Access Points (LSAPs). Each service—think of each as a data stream—is assigned a one-byte number, called an LSAP Selector (LSAP-SEL). These selectors are assigned dynamically, but are assigned from specific ranges to indicate the nature of the data connection (a service ID lookup from a directory, for example, is different from a "connected" service). LSAP-SELs are "recycled" and reassigned when the data stream that used one is terminated.

The IrDA specification demands a minimum set of services from an IrLMP implementation:

- *Device Discovery:* This service exchanges information with the other IrLMP layer to discover information about the other device.
- *Connection:* This service establishes a data connection over the IrLMP layer, setting up the service IDs and recording them.
- *Data Delivery:* This service exchanges data between the two sides.
- *Disconnection:* This service closes the IrLMP layer connection, including resetting the service ID and erasing table entries about the stream.

Service IDs and applications waiting for connections must be recorded so that data streams can be run through the same IrLAP layer and still make it to their appropriate destinations. This recording service is called the Information Access Service (IAS). IAS acts like a "yellow pages" service for a device, allowing for fixed services with pre-assigned IDs or ones that are dynamically registered with temporary IDs.

TinyTP: Tiny Transport Protocol

The Tiny Transport Protocol (TinyTP) layer is an optional layer in the IrDA stack specification. It is so important, however, that it is provided as a layer in most implementations (including Symbian OS). TinyTP provides two essential functions to the IrDA stack: flow control and data stream segmentation and reassembly.

Flow control is the most important use of TinyTP. Recall that flow control is available at the IrLAP layer, and that IrLAP sees one stream of data flowing between two devices. If one application were to require a data stream to stop briefly, the only alternative is the IrLAP layer, which would have to stop the entire stream, not just the stream for the application. The advantage that TinyTP provides is the ability to regulate data flow on a per-IrLMP-connection (i.e., per application) basis, allowing the other applications on the device to continue using the IR data stream.

The other function TinyTP performs is the segmentation and reassembly of data. Here, the TinyTP layer will receive a data stream from an upper layer, and break that stream into pieces. These pieces are numbered and then sent to the other side. The TinyTP layer on the recipient's end will then reassemble the stream based on the numbering and present the original data stream to the upper layer application. The pieces are called Service Data Units (SDUs) and the size of these SDUs is negotiated when the TinyTP connection is made (actually, the negotiation is done by the IrLMP layer and passed up).

This segmentation and reassembly is important for two reasons. First, it helps in the interleaving of various application data streams. The more pieces we have, the better we can interleave them. Second, it enables error control. In one long data stream, even a tiny error can cause the entire stream to break down and be re-sent. When that stream in broken into pieces, one error results in only one SDU re-sent. Error recovery is easier and much more efficient.

Programmer API: IrCOMM, IrOBEX, IrTranP and IrLAN

At the top of the IrDA stack are three protocols that Symbian OS ties together to form the layer that applications will interface

with. These are IrCOMM for serial and parallel port emulation, IrOBEX for data object exchange, and IrLAN for local area network attachment.

IrCOMM was designed as a "legacy" protocol; existing applications were supposed to be able to use this protocol as an emulator for standard wired serial and parallel communications. The reality is that, because IR is so different from wired communications, the IrCOMM standard requires changes in existing applications to use it. However, these changes are minimal and, for the most part, IrCOMM meets its design goals. Actually, new applications would be better served if they avoided IrCOMM and used IrOBEX or IrLAN, or even TinyTP directly. IrCOMM hides some of the useful features of the lower protocols; for example, parameter negotiation is not available (all traffic runs at 9600 bps!).

IrOBEX, or Infrared Object Exchange Protocol, is designed to allow flexible, rapid exchange of data objects in a very abstract manner. Applications that use IrOBEX are those that simply need to ship an object to the other side of the connection, with little or no concern for how it gets there. IrOBEX is a simple and compact protocol, yet it is flexible enough to handle any data object. The protocol sets up a session with the other side, negotiates an acceptable set of parameters, exchanges data objects as an abstract bit stream, then gracefully closes the session. It has some extra features that allow connections to stay alive even if data transfer is terminated prematurely.

Infrared Transport Protocol (IrTranP) is an optional protocol used to transport images from devices that can capture digital images (e.g., the Nokia 7650 phone or a digital camera). While this protocol is a standard, its limited application means that programs that use it and research work to extend it are slow to develop. Because of its specific application, it is a protocol with low overhead. Like IrOBEX, it sets up its own session with the remote imaging device, negotiates an acceptable set of parameters, exchanges images as a bit stream, then shuts the session.

IrLAN is a protocol that allows the originating device to connect to a local area network. In most uses, the "sender" in this case is a device that wishes to connect to a network; the "receiver" is an access point or adapter that is IR-enabled, and is connected directly to a network. In this mode, the receiving device becomes a gateway to the network in question. A second mode can be used with IrLAN where two devices establish a LAN between them, communicating as if the two devices were on a larger network. In either mode, IrLAN allows devices to use network protocols as if they were directly connected.

For Further Reading

IrDA protocols have an extensive set of standards defined for them. Most computers that have an infrared port—handheld or otherwise—abide by these standards. You can read more about these standards at the Infrared Data Association's Web site: ***http://www.irda.org/standards/standards.asp***.

4.2.3 PLP

Psion Link Protocol (PLP) is the protocol currently used to synchronize data between a Symbian device and a personal computer. It allows the PC to manipulate files on the attached device and manage software installation. It was developed by Psion Computers and is still used in Symbian OS.

We cover PLP in this section because it was designed to run over serial transport mechanisms, and is itself a transport protocol. It will run over serial or IR connections.

PLP is designed in three layers in addition to the physical layer, as shown in Figure 4.3. The physical layer is either the serial or the IR port. PLP uses hardware handshaking and has a frame format of 8 bits, 1 stop bit, with no parity. Any data rate supported by the physical layer can be used. Communication is asynchronous.

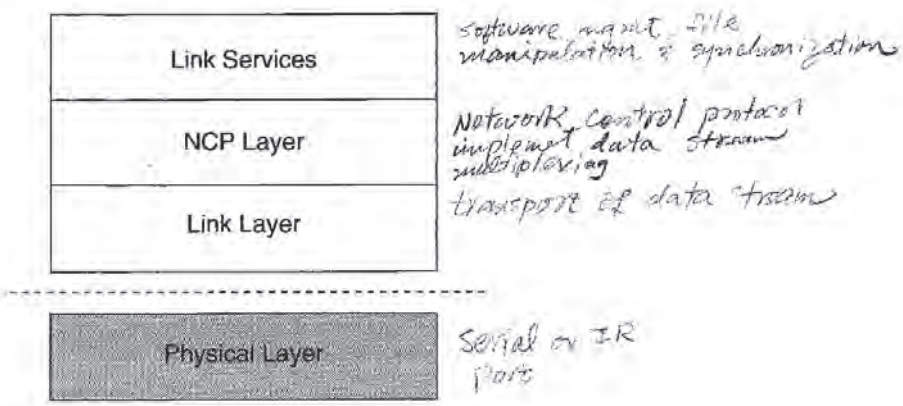

Figure 4.3 PLP protocol stack.

The link layer implements reliable transport of the data stream between the Symbian device and the PC. The data frames from the upper levels are encapsulated in a data frame with a CRC checksum at this level. Upon receipt of a data frame, the receiver recomputes a CRC checksum and compares it to the one in the data frame. It implements error recovery by implementing retransmission of data frames.

The Network Control Protocol (NCP) layer assumes a reliable transport mechanism and implements data stream multiplexing over the link layer. As with IR transmission, multiple services can be carried over one link layer data stream, and the NCP layer implements this. In addition, the NCP layer implements flow control over each service's data stream.

The final layer is the Link Services layer. This layer implements the various services used by the PC: software management, file manipulation and synchronization of files. This is accomplished as a client-server relationship, where the PC becomes the client and the Symbian device works as the server.

SyncML

PLP is a Symbian proprietary protocol. Most handheld manufacturers have their own proprietary protocols for synchronizing with desktop PCs. Because of this, each handheld device needs different software to run on a PC for synchronizing or file management.

A new standard called SyncML will help with this problem. It is a synchronizing protocol standard that most handheld manufacturers—including Symbian as a sponsor—have agreed to adopt. This would enable several things—including using one program to synchronize several devices and using networks as well as serial cables for synchronization.

SyncML is covered in more detail in Chapter 17. The Web site for SyncML technology is ***http://www.syncml.org***.

4.3 Network Technologies

As a transport technology, serial communication works well, but has several constraints. It is point-to-point only, and is *fixed* point-to-point. Once connected, two computers must (in the wired case, physically) break their connection to connect to another device. It is relatively slow. While in certain cases, IrDA communication can reach 4 Mbps, most connections attain a maximum of 115 Kbps.

Networking technologies overcome these constraints. They use multipoint communication that employs a different medium to give networked devices access to multiple machines at a time using very fast speeds. Ethernet, for example, can give access to over 2^{48} machines at one time (or 2.81×10^{14}), at speeds up to 1 Gbps.

As we examine these technologies, we need to pay attention to *addressing*. Since multiple devices can coexist on a network

medium, there must be a way to identify each one. A device is typically assigned an *address* to identify it to the rest of the network. While addressing schemes vary across networking technologies, they are usually made up of numbers arranged in a special way (significant to the networking scheme). Also, if one networking method supports another (e.g., Ethernet supporting TCP/IP), then multiple addressing methods will be used across one network.

4.3.1 Ethernet

Many networking implementations have been developed over the years, the most popular and stable of which is Ethernet. Standardized in 1981, Ethernet technology implements the data link layer from the ISO network layer diagram in Figure 2.1. Several features of the Ethernet standard characterize it.

- *Ethernet uses a bus-oriented, passive medium.* The medium that conveys Ethernet data is passive, that is, it accepts all data and broadcasts that data to all machines connected to it. As on any bus-oriented medium, all machines connected to the Ethernet can access all data broadcast over it. Each machine on the Ethernet must be well-behaved and ignore data not meant for that machine.
- *Ethernet uses a collision detect mechanism.* The method used by Ethernet to access the network medium is called Carrier Sense Multiple Access with Collision Detect (CSMA/CD). This method uses a carrier signal to convey the status of the transmission medium. When that signal implies that no one is sending information, the sending machine will send data. If there is a machine using the medium, the sender waits. If however, two machines send at the same time—i.e., a collision occurs—then both machines will detect that collision and employ a resolution algorithm to resend their data at a future time.
- *Ethernet uses special addressing.* One of the requirements of a multipoint access medium is that each machine accessing that medium needs to have a unique address. Ethernet addresses are in the form of 6 8-bit quantities. These 6 octets are typically written in hexidecimal with colons to separate them, as below:

```
12:40:bc:f2:55:10
```

- *Ethernet implements the data link layer of the protocol stack.* As with other protocol stacks we have seen, the protocol just above the physical medium—i.e., the data link layer—has some specific responsibilities. As such a protocol, Ethernet must provide

a reliable transfer of the data stream, and a method for error detection and recovery. Ethernet is specified as asynchronous, and its data frame has a sequence to start and stop it. It includes a CRC checksum on the data frame so the receiver can detect errors, and error recovery is based on retransmission of data frames.

Ethernet looks quite a bit like many of the other transport protocols we have seen in the chapter. The biggest difference is the broadcast nature of the medium and the resulting need for addresses.

Symbian OS supports Ethernet by providing drivers for various Ethernet hardware devices and the CSY module for the communication server.

Wireless Ethernet

Although Ethernet was invented as a wired technology, it has been adapted to be wireless. Three standards carry wireless Ethernet. The IEEE 802.11b standard is the most widely used at the present time. It operates at a radio frequency of 2.4 GHz and has a maximum data rate of 11 Mbps. IEEE 802.11a is a wireless standard that operates at 5.0 GHz and has a maximum data rate of 54 Mbps. Obviously, 802.11a seems more inviting. However, 802.11b has a greater reception range and is more flexible. In addition, the two protocol suites are incompatible; equipment that uses one standard will not work with equipment that uses the other.

The most recent standard tries to bridge the gap between these two. IEEE 802.11 g is a standard that works at 5.0 GHz, has the high data rate of 802.11a, and can adapt itself to work with both 802.11a and 802.11b.

More information on these protocols can be found at the IEEE WLAN Standards Working Group Web site at ***http://grouper.ieee.org/groups/802/11***.

4.3.2 TCP/IP

Ethernet implements the data link layer in the ISO stack model. Other protocols work on top of Ethernet to make a complete set of network protocols. The most widely used of these is Transmission Control Protocol / Internet Protocol or TCP/IP. IP implements the network layer in the ISO stack; TCP implements the transport layer. Both protocol suites include an implementation of a set of protocols that implement each ISO stack layer. Because of its stability and

assign port #
TCP → transport layer
IP → network layer
IP addr (network ID + host ID)

open availability, TCP/IP is the de facto standard for linking networks that use computers with a variety of different operating systems.

TCP/IP is known for the services it defines over its transport layers (e.g., Ethernet). Services like remote access via telnet, file transfer via FTP, email delivery via SMTP, and Web page services via HTTP are well known. There are many services in existence, with more added all the time. Each service at the TCP level is assigned a *port number*. On the server computer, these port numbers are connected to an application or server program. Therefore, connecting to a machine on port number 23 will probably start a telnet server program that will accept a telnet connection and will perform telnet services.

TCP/IP was originally designed to run over wired network media, to support both local area networks and wide area networks. The intent of its designers was to be a communications protocol suite that would tolerate multiple failures (its original design was for wartime communications) and would accommodate new protocols and expanded usage. Because of its flexibility, TCP/IP is an extremely stable protocol suite whose functionality has been expanded over new media and new computing innovations.

In order to link many different networks together, the IP layer needs to be able to route data frames between various networks. To allow routing, IP introduces its own set of addresses, ones that are organized in a manner that will assign machines on a local network addresses from the same address group. So the IP layer makes addresses that are four integers (8 bit quantities, less than 256) separated by periods:

```
192.168.112.32
```

IP addresses have a *network* ID portion and a *host ID* portion. The network ID is used to identify the network to which the address belongs; the host ID identifies the computer on its home network (the one given by the network ID). The IP address is broken down into these two pieces by a *subnet mask* that is ANDed bitwise with the address to reveal the network ID. For example, a subnet mask of `255.255.255.0` will extract a network ID of `192.168.112.0` from the above address.

In addition to a new form of addressing, the IP layer implements a new form of datagram exchange between machines. It uses its own form of datagrams, separate from whatever format datagram is used by the data link protocol that supports it.

TCP is mainly concerned with adding reliability to the transport of the application data stream. As is done with other protocol stacks, TCP breaks the data stream into pieces, building its own datagram from each piece. Each datagram is numbered and is sent to the IP layer with a CRC checksum. Once they arrive at their destination, the datagrams are checked for errors and reassembled.

In addition to these features, TCP adds the idea of *connected sessions* between machines. Data exchange sessions are data exchanges that remain active for the exchange of many datagrams at a time, without the need to reopen or reconnect the data path between two machines. This results in a more reliable interface that is easier to program.

Two examples of the flexibility of TCP/IP are Point-to-Point Protocol (PPP) and IrLAN. PPP is a data link layer protocol that runs underneath TCP/IP and supports its operation (replacing Ethernet). PPP is implemented across two machines, building a data link layer in the point-to-point manner (hence the name). While PPP has been implemented for many point-to-point situations, its most effective use has been in hooking up dial-up users to the Internet. A typical dial-up scenario is shown in Figure 4.4. Here, a user at home dials up an Internet Service Provider (ISP) via a modem. An access server at the ISP authenticates the user and gives the user access to the ISP's local network by assigning the dial-up computer an IP address for duration of the phone call. The ISP's local network has a gateway to its Internet connection, and that gives the dial-up computer access to the Internet. When the dialup connection is severed—either by the user (hanging up) or the ISP—the PPP connection is terminated and the IP address is recycled and given to another dial-up user.

PPP
dial-up users to the Internet.

PPP provides three main features:

- *Option negotiation:* As with other protocols we have seen, PPP engages in a series of negotiations with the access server that grants access to its local network before completely establishing a data link layer connection. Configuration options such as an IP address, error correction methods, and framing protocols are decided in these negotiations.
- *Framing method:* PPP uses asynchronous communication, and because of this, establishes a special framing method. This framing method looks much like those we have seen before. It uses a special character sequence to start and end frames, and includes error control information (including CRC checksums).

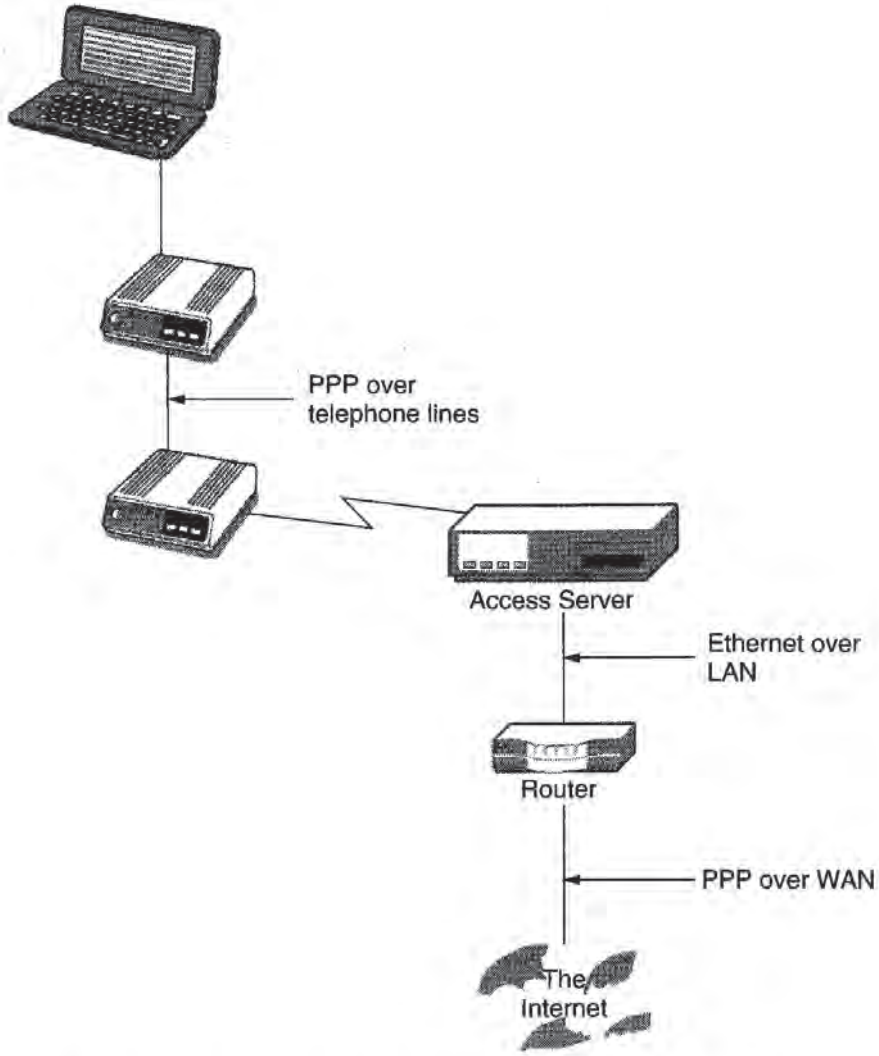

Figure 4.4 An example of using PPP to connect to the internet.

- *Link control protocol:* PPP implements the data link layer across its asynchronous connection, including: data frame transport, measurements of line status (e.g., if a connection is up or down), and frame error detection and correction.

Symbian OS has complete support for PPP, most often used for mobile dialup to the Internet.

IrLAN is another data link layer protocol that supports TCP/IP. We have already discussed IrLAN briefly. Basically, IrLAN looks very much like PPP; they both use configuration negotiation, they construct similar data frames, and they implement a similar link control protocol. Instead of modems, IrLAN uses a network access adapter that has an infrared port and is connected directly to a LAN. The IrLAN protocol works between the computing device and the IR network access point.

Snooping

Networks are notorious security holes. One side effect to the open standards of Ethernet and TCP/IP is that it is easy to discover and decode data frames. And because Ethernet implements a passive medium, network traffic is easy to snoop without detection. This means that these networks are intrinsically insecure and more steps need to be taken if data is to be protected from prying eyes. Chapter 6 deals with security and with snooping in various forms.

4.4 Wireless Access Protocol

It is tempting to view Wireless Access Protocol (WAP) as exclusively a content technology. It is, after all, focused on content for small devices. However, underlying the content technologies is a transport and delivery platform that we should overview in this chapter. The WAP architecture is shown in Figure 4.5.

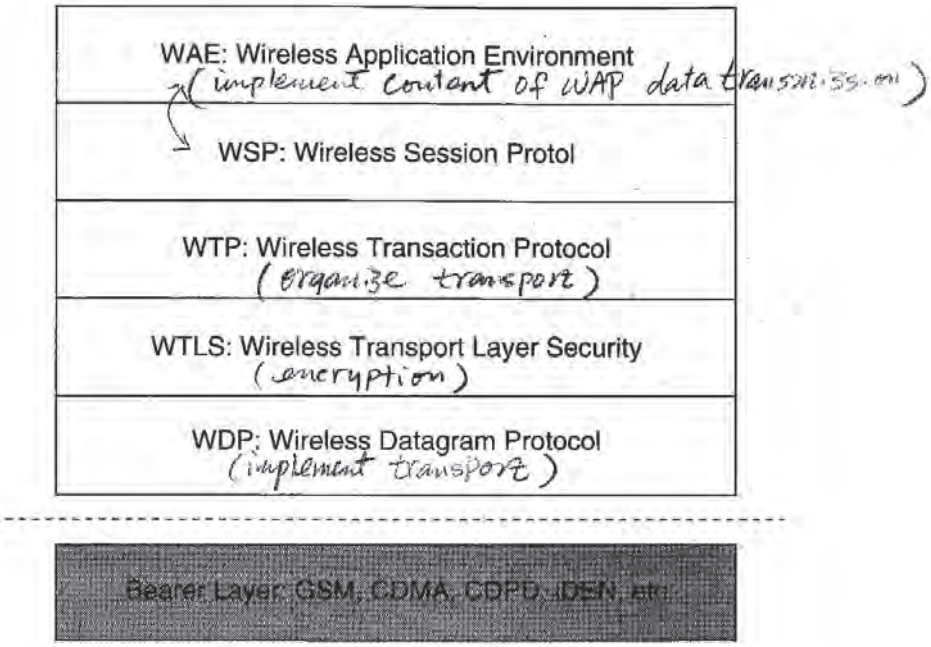

Figure 4.5 The WAP stack.

At the base of the WAP stack is the bearer layer. This corresponds to the physical layer in other protocol stacks but is not specified for WAP. The designers of WAP assume that there will be "bearers" that will support the upper WAP structure and will be able to

interact with the upper structure through a specified API. This bearer layer can be implemented by any digital carrier, and is currently implemented over many mobile phone protocols, including GSM, which is supported by Symbian OS.

The Wireless Datagram Protocol (WDP) layer provides network and certain transport functions to the rest of the WAP stack. It provides the upper layers with datagram transport between stacks, implementing port level addressing. It provides for datagram segmentation and reassembly, supporting selective retransmission of datagrams, and for error recovery.

The Wireless Transaction Protocol (WTP) layer provides the rest of the transport layer functions. It provides both connection-oriented and connectionless transactions between WTP layers that use the segmentation and reassembly functions of the WDP layer. It directs the retransmission of datagrams and can concatenate data packets into larger packets for efficiency.

The Wireless Transport Layer Security (WTLS) layer lies between the WDP and WTP layers. This layer provides encryption of data between client and server and provides methods for authentication. It ensures the integrity of the data exchanged. It exploits the modularity of the protocol stack model to provide upper layers with unencrypted data, while lower layers see only an encrypted data stream. It has facilities for detecting when data is sent repeatedly or unverified (e.g., attacks on system security) and can reject data in these cases.

Note how the transport functions in the WAP stack have been split between the WDP and WTP layers. This is to facilitate the insertion of the WTLS layer. For example, the WTP layer will coordinate packet segmentation, but the WDP layer will implement the segmentation *after* the data packet has been encrypted. In general, the WTP layer will organize the transport and the WDP layer will implement the transport so that the WTLS layer can do its work.

The top layers in the WAP stack—the Wireless Session Protocol (WSP) layer and Wireless Application Environment (WAE) layer—implement the content of WAP data transmission. We will overview these layers in the next chapter.

4.5 Bluetooth

We have discussed several kinds of connections between machines—from point-to-point connections to wide area multipoint

connections—using cables, infrared, and radio communication. The focus of these connections has been to connect independent computing devices to each other, where clients take advantage of services offered by servers. Also, we have specifically made the concept of *distance* to be irrelevant; as long as there are enough computers that will link networks together—either wired or mobile—we can dismiss the distance between two computers.

The Bluetooth model of connectivity zeroes in on these two areas: it allows *components* as well as independent computers to use wireless connectivity, and it specifically makes distance a factor, focusing on short range wireless connectivity. For example, a headset can be connected to a mobile phone via Bluetooth. The headset is designed as a component of a mobile phone system (i.e., it cannot function on its own) and will not work when it is separated by more than 10 meters from the phone. By broadening the possibilities of connections and restricting those connections to short distances, Bluetooth makes many new communications innovations possible. In contrast to wide area networks (a regional network called a WAN) and local area networks (often restricted to a building), it establishes the idea of a "personal area network" (PAN), also called a "piconet". In a piconet, a network of computers and components connect to each other over a small (personal) area. Bluetooth also addresses the connection of piconets to each other, forming an ad-hoc "scatternet" that communicates over network gateways. PAN

Consider a few Bluetooth scenarios. A piconet could be a computer system, where the components are connected to each other without cables. A mouse, a keyboard, and a printer could all be Bluetooth enabled and work together. A wireless phone that senses its proximity to a base station could dynamically choose to use landline or mobile communication. A headset could be wireless and connected to a mobile phone stored in a briefcase.

The Bluetooth model spans the layers of our ISO stack model from Figure 2.1. The model reuses existing protocols as much as possible and substitutes only those layers that are necessary for radio transmission and accompanying management needs. Therefore, protocols such as PPP and OBEX can run over Bluetooth protocols, and Symbian devices implementing Bluetooth can reuse many protocol modules.

The Bluetooth protocol stack is given in Figure 4.6. Symbian OS supports this protocol stack from version 6.1. We will discuss each of the components in turn.

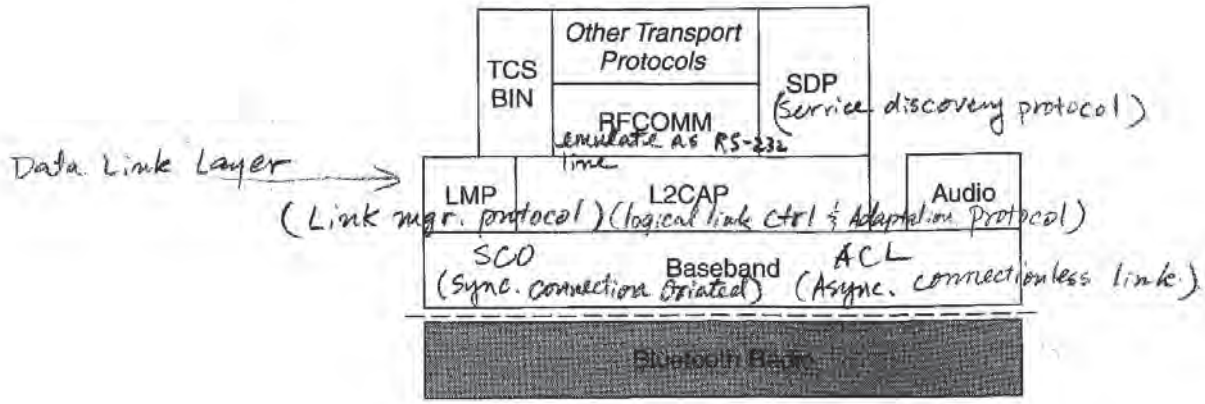

Figure 4.6 The Bluetooth stack.

The Bluetooth Consortium

As a wireless standard, Bluetooth is one of the most endorsed protocol suites that has been proposed in recent years. The Bluetooth Consortium is the standards group, and its list of members has about 2500 companies. Membership is free and open to any incorporated company. The membership is very broad, from semiconductor manufacturers to telecom companies to operating system developers. Since membership gives a company a free license to Bluetooth implementations, there is a great incentive to join.

There is great commercial value to building a consortium with many members. As more companies join the consortium, each has free access to Bluetooth specifications, standards, and reference implementations. In turn, this helps each manufacturer to implement Bluetooth technology for itself, which spreads that technology around the computer industry and helps bring Bluetooth to market. More information can be found at ***http://www.bluetooth.com.***

4.5.1 Bluetooth Radio

This layer is the physical layer of the Bluetooth stack. Bluetooth uses radio frequencies in the Industrial, Scientific, and Medical (ISM) band allocated at 2.4 GHz. This ISM band is unregulated, therefore no license is needed to operate in it (actually, the ISM band varies a bit from country to country, so Bluetooth takes care to stay out of the way of applications that use out-of-ISM frequencies). Bluetooth is designed to operate in very low power situations; the specifications give power requirements from 1 mW to 100 mW.

This gives components the ability to send and receive Bluetooth signals at low power. This also helps manufacturers keep both the size of the Bluetooth chip and the power supply to the chip small.

Further Reading on Bluetooth Radio

Further details about the Bluetooth Radio layer are beyond the scope of this book, but you can read more about the specifications on the Web site of the Bluetooth Consortium. Technical specifications on the Bluetooth standard—including information on the radio layer—can be found at ***http://www.bluetooth.com/developer/specification.core.asp***.

4.5.2 Baseband

Baseband serves as the software complement to the radio hardware and straddles the physical and data link layers. The services at the baseband layer regulate the radio hardware so that the device abides by the Bluetooth standards. Data encoding, transmitter management, frequency hopping, security, error detection and correction, and framing are all covered by the services specified at the baseband layer.

The baseband layer uses a frequency hopping, spread spectrum system. Data frames are transmitted in defined time slots through synchronized transmission. By using a combination of several frequencies and a wide range of hopping sequences, Bluetooth transmissions can be reliable and can avoid collisions with transmissions on physically adjacent piconets.

The baseband layer implements two types of link. Synchronous Connection-Oriented (SCO) links are made on a point-to-point basis between one primary and one secondary device. These links are used to acquire fixed, reserved allocations of bandwidth between the two devices involved in the link. Through this reserved bandwidth, the baseband layer transmits audio or a mix of audio and data (see Section 4.5.7 below for Bluetooth audio). Asynchronous Connectionless (ACL) links are used in a more flexible fashion. They can implement point-to-multipoint connections and are asynchronous. ACL links are used for data only.

The Bluetooth standard requires that up to three SCO connections can be transported in a single piconet, and ACL traffic is limited primarily by the transmission quality and data rate of the underlying RF transport. The ACL traffic flow is managed by the primary station, with secondary stations speaking only when spoken to.

The baseband layer is usually provided as a hardware implementation. It accompanies the radio transmitter chip on a Bluetooth card.

4.5.3 LMP: Link Manager Protocol

The Link Manager Protocol (LMP) is used to establish, secure, and control the links between devices. LMP is defined at the data link layer.

LMP is not used by upper layers. It provides the establishment of links, which get handed off to the L2CAP layer. The devices involved in the link are authenticated first, and the data streams may be encrypted. LMP also controls transmission power on the Bluetooth device, sometimes at the request of the remote host (a device may request that the connected device increase or decrease its transmission power based on the signal strength from that device). In addition, this layer is responsible for monitoring the link between devices and watching for link quality.

4.5.4 L2CAP: Logical Link Control and Adaptation Protocol

The Logical Link Control and Adaptation Protocol (L2CAP) Layer is also a data link layer protocol, implemented alongside the LMP. The L2CAP layer provides the following services to upper protocol layers:

- *Connectionless and connection-oriented data services:* The L2CAP layer connects upper layers to the baseband layer through two types of connections. Connectionless sessions send data frames that are not tied to each other; connection-oriented sessions send data frames that are sequenced and numbered, and that must be delivered in order. The L2CAP layer provides reliability through connection-oriented service.
- *Protocol multiplexing:* Because the baseband layer does not distinguish between data streams from separate upper layers, the L2CAP layer must multiplex between the various types of upper layer protocols. It gathers all upper protocol types and presents one data stream to the baseband layer.
- *Data stream segmentation and reassembly:* This function provides the freedom for upper layers to make packets that are larger than the baseband layer will handle. L2CAP makes this possible by segmenting large upper layer data frames into frames that the baseband can handle. L2CAP also reassembles frames from the baseband into the appropriate frames for upper levels.

This allows upper layer flexibility and efficiency in protocol interleaving.

- *Group abstractions:* L2CAP manages protocol groups, allowing a mapping of the protocols onto the piconet concept defined by the baseband. This is an abstraction of a multicast. Protocols from upper layers may want to address all devices in the local network. Since upper layers do not know which devices are in the local network at any given time, they rely on the L2CAP layer to keep track of this. The upper layer will refer to a group of devices; the L2CAP layer will map this group into the currently defined piconet.

The L2CAP layer is defined only through ACL connections to the baseband layer, and therefore is used only for data transmissions.

4.5.5 SDP: Service Discovery Protocol

Bluetooth devices operate by participating in a Bluetooth piconet or scatternet. Maintaining a stable list of these services can be quite difficult, however, given the dynamic nature of Bluetooth networks. So a Bluetooth device must *discover* the services that are available instead of being configured with a static list of them. Service Discovery Protocol (SDP) allows a Bluetooth device to do this.

SDP allows Bluetooth devices to *discover* services, to *search for* services, and to *browse* services. This applies to new devices joining a piconet and for devices already part of a piconet that are exposed to newly offered services. We can think of this like a client in a client-server model; the device offering the service is like an SDP server and the devices looking for or discovering the service are like SDP clients. We should also note here that SDP is a distributed protocol. When a service discovery needs to be made, there is no central place where this information is stored. Rather, SDP must broadcast queries and field the responses to find out which device offers specific services.

So as to be transport independent, SDP is implemented on top of the L2CAP layer.

4.5.6 RFCOMM

The RFCOMM layer is not a "core" Bluetooth layer. Rather, it is a layer built on top of the core layers to allow an interface between existing services and the Bluetooth stack. RFCOMM emulates an RS-232 serial line.

To upper layers of existing protocols, e.g., OBEX, RFCOMM presents an interface that looks exactly like an RS-232 connection (remember IrCOMM?). It provides all the transport capabilities that upper level services expect. It interfaces with the L2CAP layer and implements a point-to-point transport method. Up to 60 connections can be made between devices.

4.5.7 Other Stack Components

There are a few other components of the Bluetooth stack that warrant attention.

Telephone Control Protocol—Binary (TCS BIN) is an optional bit-oriented, call control protocol that enables Bluetooth devices to establish speech and data "calls" with other Bluetooth devices. The idea here is to allow Bluetooth-enabled mobile devices to make calls to each other—both voice and data. TCS BIN enables devices to make calls between two devices or from one device to many devices. It also allows connectionless calls, one-time messages or data drops to one device or a group of devices. TCS BIN works above the LMP and L2CAP layer.

There is a set of "AT" commands defined for Bluetooth. "AT" commands are used to give control commands to modems. The specification of Bluetooth "AT" commands defines how Bluetooth devices could fit into a standard computer-to-modem scenario. It is an attempt to allow Bluetooth devices to work with legacy equipment and software. The "AT" command set is defined for Bluetooth mobile phones and modems, and works for data calls as well as fax calls.

Finally, a note should be made about Bluetooth audio. Figure 4.6 shows that Bluetooth audio bypasses LMP and L2CAP and interfaces directly with the baseband layer. Audio data is transferred between Bluetooth devices as raw data routed directly to and from the baseband layer. The Bluetooth audio model is very simple; while devices exchange data tagged as audio data by opening a special link, each application is free to manipulate the data as it sees fit.

4.6 Telephony

The last communications technology that we will cover in this chapter is telephony. Loosely defined, telephony is the technology usually associated with the transmission of voice, traditionally associated with telephone communication. It is, however, a

point-to-point communication medium that can be viewed in much the same way we view other data networks. Although voice transmission was the first—and is still the foremost—use of telephony, data can also be exchanged using most telephony methods. In fact, the units that use telephony can be addressed by their phone number.

We can use two dimensions through which to view telephony. The first way to view telephony is in two broad categories: analog and digital services. Analog services are those provided using traditional telephone equipment, which alters the telephone line signal's amplitude or frequency to reflect the transmitted voice quality or intensity. Digital services are those provided using media that can transmit the bit states of digital data—1s and 0s provided by changing the state of the signal into one of two states (or three, using the third state to indicate a carrier signal).

A modem provides a good example of the difference between these two categories. When a computer transmits a data frame of bits to another computer, each side expects the data as digital data. However, if telephone lines are between them, then the data must be transferred in an analog fashion. A modem is a digital-to-analog converter, taking the digital data from a computer, and converting it to sound for transmission over telephone lines. The sound has three distinct types, each representing a digital state. On the receiver side, the modem "hears" the sounds on the telephone line, and converts it to bits for the receiving computer.

The second dimension focuses on the telephony signal and defines wired and wireless signal carriers. These categories need little elaboration; wired telephony has a long history while wireless telephony is a more recent invention.

Figure 4.7 provides a good look at how these two dimensions interact. For example, traditional telephone service has been carried over wires. The interface between the computer and the phone line has taken on much of the terminology of a human using a telephone. Each analog line has two states: on-hook, which describes a disengaged line (think of the telephone handset as on its cradle or hook), and off-hook, which describes an engaged line (with the handset removed from its hook). Computers connect to each other by dialing phone numbers—sometimes referred to as "addresses"—and interpreting responses humans usually get, such as busy tones or phone ring tones. A computer should be aware of services the telephone service provider puts on a phone line, such as call-waiting or call-forwarding. These are configured by hand for each phone line.

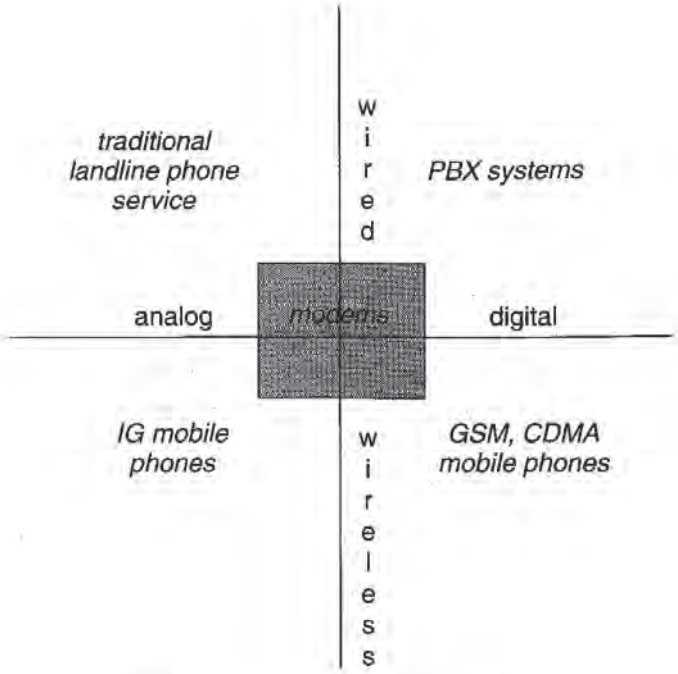

Figure 4.7 The two dimensions of telephony.

A mobile GSM phone is a good example of digital wireless. Just as we have seen in other digital implementations, GSM is based on data packets in a specific format. GSM uses time division multiplexing (TDM), which is a way to have up to 8 separate connections to one device over any of 124 frequencies at any one moment. Sharing that device is based on synchronous communication and depends on fixed-time slots in which 148-bit data frames are sent. The data rate can be high—32,500 bits are sent every 120 milliseconds—but the overhead on each data frame is also very high. Because it is digital, a GSM phone can receive messages as well as phone calls, and can act as both a telephone and a modem. As a telephone, GSM phones convert analog voice to digital data. Because of aspects like high overhead, the modem data rate is not fast, typically a maximum of 19,200 baud. This will change soon, however, as new protocols and faster transmission technology is implemented.

Notice in Figure 4.7 that modems span all four quadrants. Modems can be connected to analog, wired phone lines as well as wireless digital lines. When used with digital technology, the actual "modem" functions—the modulation and demodulation of the digital signal—are blurred. The connection of one computer to another through a phone service is still the primary use and so the "modem" term sticks.

4.7 Summary

This has been a swift (but not brief), rather sweeping survey of the transport technologies supported by Symbian OS. We should take a moment to get our bearings and re-examine the protocols. Figure 4.8 contains a good diagram comparing the various protocol stacks. The comparison is to the ISO stack model, shown in Figure 2.1.

Serial transport methods address a portion of the data link layer. Implementations of serial transport implement enough of the data link layer to let applications at upper layers talk to each other. There is no reliability and very little error correction. The maximum speed of this technology is 115 Kbps, using primarily asynchronous communication.

IrDA-based transport methods are the most comprehensive and most specific. They specify implementations from the physical layer up through the session layer and work through tight integration of all protocols. The top asynchronous speed of this protocol set is 115 Kbps; a top speed of 4 Mbps can be reached using synchronous implementations.

PLP methods are synchronization protocols. These protocols implement only the data link and network layers, but offer very effective services as applications on top of these two layers. Because PLP is implemented on top of asynchronous serial transport—either cabled serial or IrDA methods—the top speed of PLP is 115 Kbps.

The networking and TCP/IP suite of transport methods implements a very complete set of protocols. While it does not specify a physical layer, its various data link layer implementations require specific physical layer connections. It offers many different protocols and services from the data link layer up through applications. This suite implements what most would consider to be network protocols. Speeds for TCP/IP can range from 115 Kbps for serial PPP to 10 Mbps and 1 Gbps for Ethernet.

The WAP protocol stack mimics TCP/IP, but customizes it for small devices and mobile phones. It lacks both physical layer and data link layer specifications, assuming that bearers—that is, device manufacturers—will provide these layers in hardware or firmware. There is an additional security layer added for security that provides data encryption, authentication, and denial of service detection. Speeds for WAP depend on the speed of the bearer, currently not over 38.4 Kbps.

Bluetooth transport methods are the most recently developed methods and, like IrDA specifications, are tailored for a specific

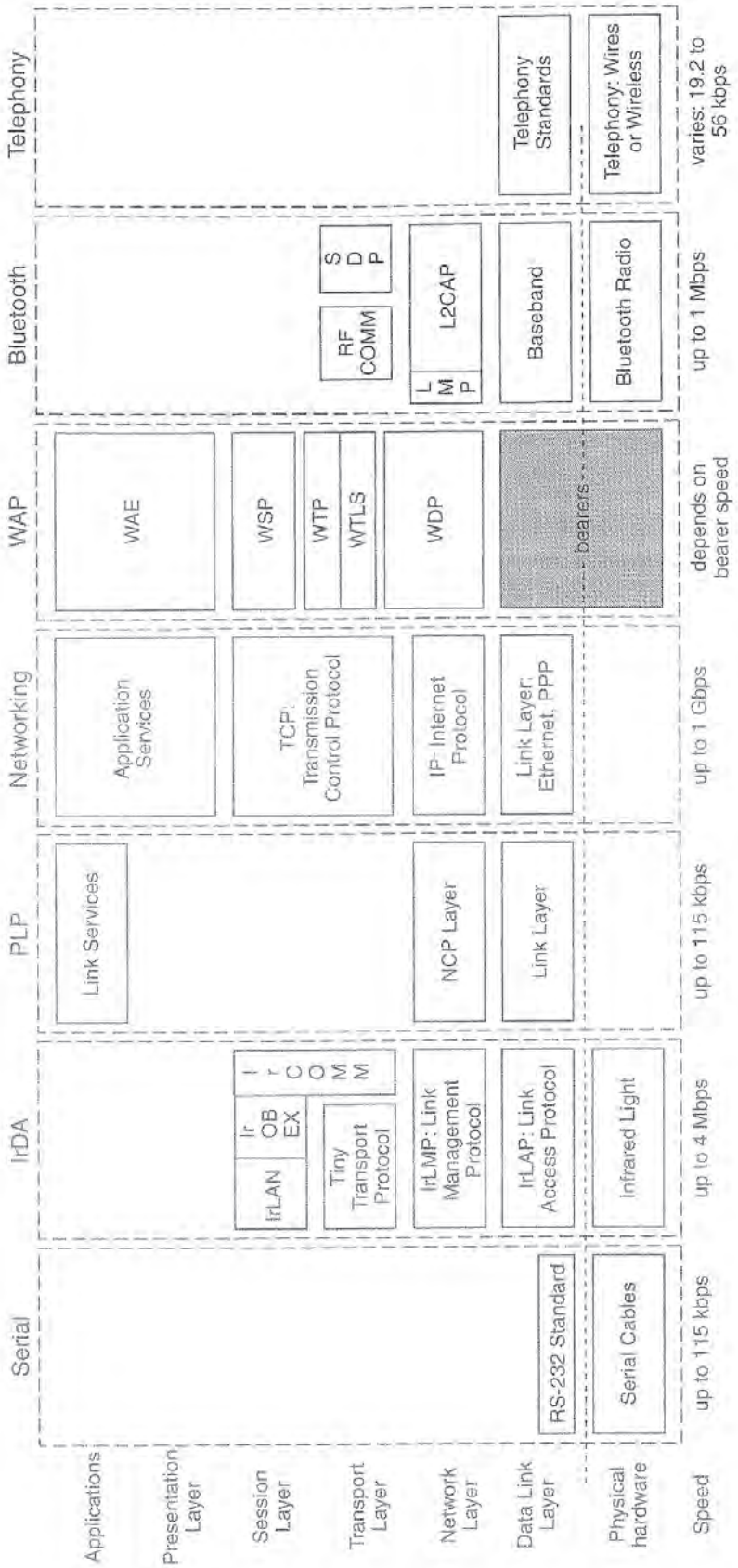

Figure 4.8 Comparing the protocol stacks.

physical medium. Bluetooth specifies the physical layer, and tailors the rest of its stack to work with this physical layer. Speeds of Bluetooth methods range up to 1 Mbps.

Telephony transport methods are much like serial methods. They typically comprise only the data link layer, and allow applications to be connected to each other with no other promises. The relay of data is unreliable, and error detection is minimal. Speeds are comparable to serial communication, and can range up to 56 Kbps, depending on the implementation.

5

A Look at Content Technologies

In the last chapter, we overviewed many different transport technologies. From RS-232 serial communication to Bluetooth protocols, the common thread that drew these technologies together was that each was used to transport data between Symbian OS devices. In this chapter, we take a look at the kinds of content that Symbian OS supports in the data that is transported. We will overview content technologies along with the content itself.

Content is, after all, the most important aspect of communication. It is very important to understand the transport methods used, but once you have spent hours poring over details about those transport methods, those details fade into the background as we discuss content issues like messaging and how to use WAP. It is very easy to take the transport for granted as we start to depend on the content.

In this chapter, we will overview messaging on Symbian OS in its many forms: email, SMS, fax, and BIO messages. We will also review HTTP and WAP, two of the important content-providing technologies.

5.1 Messaging

Messages are self-contained data objects sent between two devices. They are *self-contained* in the sense that they do not depend on the sender's or the receiver's environment. They are *data objects* because they may take one of a number of forms, and their definition is open-ended. Messages are typically used to relay specific pieces of information between machines, as well as humans.

Messages have several common characteristics:

- *A sender:* Each message originates from somewhere, either from a person or a computing device. The identity of the sender is usually included with the message, although that identity can

rarely be trusted. Messages can be original person-to-person communications, or can be generated by a computer.

- *An intended destination:* Although this may seem obvious, a message is always sent *somewhere*. Messages can be sent to a single destination, to a group, or to any device that will accept it. At some layer in the transport system, messages are always sent device-to-device. The final destination, however, can be a human reader or an application.
- *Timestamps:* Messages are typically timestamped, that is, given time and date information about when an operation is performed on it. A message can receive many timestamps, and can have many operations performed on it. An email message, for example, may travel through several relay points before arriving at its final destination, and will then bear timestamps from each relay point.
- *Content:* A message always carries information to its intended destination. While that content could be empty, it is still considered to be part of the message's definition.
- *Format:* Naturally, different types of messages take different forms. However, messages tend to have a common organizational format: a *header* and a *body*. The message header contains information about the message itself, such as sender, destination, delivery options, timestamps, and the like. The body of a message contains its content, that is, the information the message was meant to convey to its recipient. Beyond this general structure, messages vary widely in how they represent or format each message section.

Consider the example found in Figure 5.1. This figure displays an email message in the form that gets exchanged between two devices (not necessarily the final form a person might read). This message has a typical format. The header is separated from the body by a blank line (about two-thirds the way down the message). You can easily spot the sender and destination by the "From:" and "To:" lines. The header contains several timestamps; the "Received:" fields display information about the relays the message went through. Note that there is a lot of information in the header, more information than is typically useful. In this case, the body of the message is an ASCII-based textual message.

Message sending and receiving typically involves both servers and clients. A typical scenario is depicted in Figure 5.2. The sender composes the message on his or her local device. The message is then "sent"—which means it is uploaded to a server, either over

```
Received: from mail.brainshareproject.com (mail.brainshareproject.com)
      by smaug.cs.hope.edu (8.9.3+Sun/8.9.1) with ESMTP id AAA09235
      for <jipping@cs.hope.edu>; Wed, 27 Jun 2001 00:33:11 -0400 (EDT)
Received: from debian (dialup-209.Dial1.Level10.net [192.245.239.242])
      by mail.brainshareproject.com (EL-8_9_3_3/8.9.3) with ESMTP
      id VAA14057
      for <jipping@cs.hope.edu>; Tue, 26 Jun 2001 21:32:46 -0700 (PDT)
Received: from jjones by debian with local (Exim 3.22 #1 (Debian))
      id 15F70M-0000UR-00
      for <jipping@cs.hope.edu>; Tue, 26 Jun 2001 23:32:42 -0500
Date: Tue, 26 Jun 2001 23:32:42 -0500
From: John Jones <jjones@brainshareproject.com>
To: jipping@cs.hope.edu
Subject: Infomercials.org
Message-ID: <20010626233241.A1876@brainshareproject.com>
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Disposition: inline
User-Agent: Mutt/1.3.18i
X-StarTrek-Quote: Make it so.
Sender: John Jones <jjones@brainshareproject.com>
Content-Length: 701

I want to bounce this site off of you:
      http://www.infomercials.org/
and hear any opinions you have on the material it presents.  They seem
to have a very well-thought-out approach to managing large (or small)
numbers of customers.

John J.
```

Figure 5.1 An example email message.

a network or wirelessly. This server, known as a message center or relay server, must now deliver the message.

There are also delivery methods that work on a peer-to-peer basis. These methods allow local message composition and deliver the message directly to the recipient. The message relay server is removed from the loop.

In general, then, there are two models for mail delivery:

- *The "push" model:* If it can, the server will deliver the message directly. This means that it must contact the destination device and "push" the message to it. This requires that the destination is powered on and can receive messages.
- *The "pull" model:* When a device is not usually connected to a network or mobile service, the server will store the message for the recipient. The receiving device then must contact the server and "pull" its messages from the server. Often, in this model, the message relay point is not the message storage point. The storage server receives the message from the message relay and keeps it for the recipient.

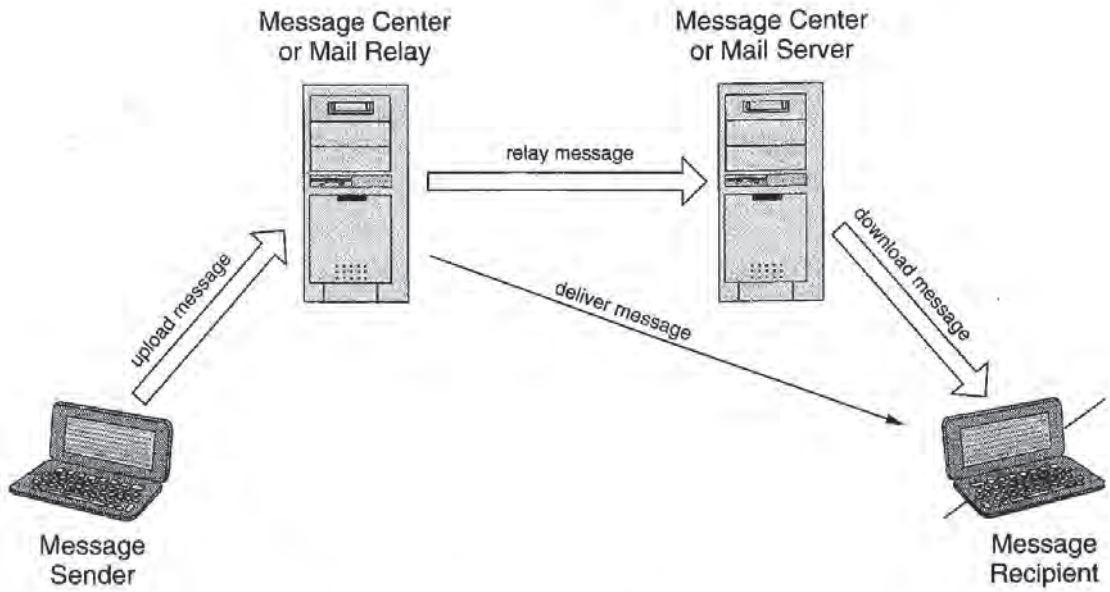

Figure 5.2 A typical message delivery scenario.

Note that the pull model could involve the push model. A device might pull its messages by simply notifying a server that it is online and ready. The server then uses push model mechanisms to deliver content to that device.

The Symbian platform supports four specific message types as well as the ability to extend the platform to embrace new ones. It also supports both models of message delivery. We will cover the message types in this section, as well as the technologies that support them. We will cover message delivery later when we discuss programming with messages.

5.1.1 Electronic Mail

Electronic mail is one of the most common and widely used forms of messaging. It was originally developed as an electronic means to send text-based messages—textual "mail" for human consumption—between computers. As the value of sending messages was realized, people began to send other things as well. Email has evolved to encompass all types of objects, for example, programs, spreadsheets, or word processing documents.

The format of an email message is exemplified in Figure 5.1.* As we have stated, email messages have a header and a body. The

* The most widely used format for email messages is specified by the Internet Engineering Task Force by a document called RFC 821.

header is comprised of a sequence of lines or fields, each of which is composed of a key and a value, separated by a colon. Each field relays information about the message to the receiver. The sender is required to insert a "From" field, a "To" field, and a "Posted-Date" field into the message header, and is free to insert other defined fields. In addition, any field whose key value begins with the string "X-" may be inserted by the user (note the "`X-StarTrek-Quote`" field in Figure 5.1).

The body of an email message contains the message content. This content is typically composed of a message in ASCII or UniCode text. However, the message body can also have *attachments*, which are data objects that accompany the message. These objects are included using a standard called Multipurpose Internet Mail Extensions (MIME). Objects included using MIME are each included in a message type format, with a header to identify the type of the data object and a body that contains the data object itself. Many objects can be included in a message.

Even when it includes MIME objects, an email message has a text-based representation. All email is sent using readable (ASCII/UniCode) characters. If a message contains attachments that are not comprised of text, then those attachments are encoded in a special way to derive text from them. The encoding method used, e.g., base 64, is included in the header for the data object, so the receiving software can decode the object.

The delivery of email messages to the end user follows the "pull" model of mail delivery.

There are several protocols that are used to send and receive email. By far, the most widely used sending protocol is Simple Mail Transfer Protocol (SMTP). There are two protocols used to receive or read email: Post Office Protocol (POP) and Internet Message Access Protocol (IMAP). These are TCP/IP, network-based protocols.

Look at Your Email

Look at your own email messages to verify the format I specify here. Save your email to a file and examine the file's contents. If you can, find a mail message with attachments and look at the contents. You may have to specify a "save headers" or "save all headers" property to your mail reader; often mail readers will save only those header fields that it deems interesting. Use these headers to track the path your email message has gone through to reach you.

SMTP: Simple Mail Transfer Protocol

SMTP is a protocol that is used to upload email from applications, and to exchange email between servers. Message senders will use SMTP when exchanging messages with relays or storage servers and relays and storage servers will use SMTP when exchanging messages between them. As its name implies, SMTP is a simple, textual protocol.

SMTP has three phases. The first is an *identification phase*. In this phase, the sender sends an "HELO" message, which includes its name. The receiver accepts this name and both sides move to the exchange phase.

In the *exchange phase*, the sender can make some general queries or can send messages to the mail server. Address verification is a good example of a general query. Sending a "VRFY" message with an email address will cause the mail server to send back an acknowledgement or an error message that tells the sender if the email address is valid on the server's system. Mail exchange is initiated by the "DATA" message and terminated by a line with only a "." character on it.

The final phase is a *termination phase*. This happens when the sender issues a "QUIT" message.

You can see how simple this protocol is. It is easily understandable and easily implemented. It is also easily broken and can be a major security risk. There is no authentication and verification of identification. Those who snoop networks can easily read and figure out what email is being exchanged.

Experiment with SMTP

It is easy to experiment with SMTP because it is a simple protocol. Identify an SMTP server—perhaps the one that receives your email. Follow these steps:

1. Telnet to this server on port 25. Use a command line like:

```
telnet smtp.server.com 25
```

2. You should get a response from the connection that identifies the SMTP server, the version of the server software, and the current date and time. Your response should be to identify yourself by giving the HELO command, like this:

```
HELO my.computer.com
```

   You give your computer name—fully qualified with a domain.

3. The response should be a greeting ("pleased to meet you"). You are then free to enter SMTP exchange commands. One easy one is to use "VRFY" to verify an email address:

```
vrfy jipping@cs.hope.edu
```

Experiment with other commands. With many SMTP servers, you can issue a "HELP" command, and the server will give you assistance for using the commands that it accepts.

4. Terminate the connection with the server by using the "QUIT" command when you are done.

POP3: Post Office Protocol

SMTP is a protocol for mail uploading and exchanging, but applications do not use it for mail receipt. Post Office Protocol Version 3 (POP3) is a popular protocol used to receive and download messages.

Like SMTP, POP3 has three phases: authorization, transaction, and termination. The *identification phase* uses a username and a password, which is verified by the mail server. This means that the receiver typically has an account on the server's system, so that the username and password can be verified. Once verification has been achieved, both sides move to the message delivery, or transaction, phase.

The *transaction phase* allows some information and maintenance messages, and allows for message download. Information messages such as how many email messages are waiting to be picked up or the size of those messages are very useful. Maintenance messages include the ability to delete messages from the server's storage. Download messages allow the receiver to download only message headers or entire messages.

Termination is initiated by the sender through the "QUIT" command. The protocol connection between client and server is severed by this command.

The POP3 protocol was designed for situations where messages are delivered to the receiver client. It is assumed that clients will manipulate and store messages on their own devices and not on the server and the default delivery mode is to delete messages from the storage server after delivery. However, most email receiver clients offer the option of leaving copies of email messages on the mail server in the user's mailbox.

POP3 uses authentication while SMTP does not. We have assailed SMTP for being insecure, but the use of authentication

does not make POP3 much better here than SMTP. POP3 still uses plain and unencrypted text characters for communication between server and client. This means that the username and password are available to snoopers and that each email that is downloaded is potentially readable by those who are watching the network traffic.

Experiment with POP3

You can experiment with POP3 very easily. Identify a POP3 server. This could easily be the one you use to download email from. Follow these steps:

1. Telnet to this server on port 110. Use a command line something like this:

   ```
   telnet pop.mailserver.com 110
   ```

2. You should get an identifying response from the mail server. You then identify yourself like this:

   ```
   user jipping

   pass jipspassword
   ```

   Substitute your username for "jipping" and your password for "jipspassword".

3. Once you have passed the authorization stage into the transaction stage, you can play with POP3 commands. Commands like LIST and LIST <msg#> will list all messages and a specific message, respectively, by their size in bytes. You can read a message by using the RETR <msg#> command and can delete messages with the DELE <msg#> command.

4. When you are done using the POP3 session, you can quit the session by issuing the QUIT command.

IMAP4: Internet Message Access Protocol

Internet Message Access Protocol Version 4 (IMAP4) is a mail download protocol very much like POP3. There are four states with IMAP4: non-authenticated, authenticated, selection, and termination.

The *non-authenticated state* is the initial state a client enters when it contacts an IMAP4 server. This is where the client will

identify itself with the username and password of its user. Upon verification, both sides enter the authenticated state.

When it enters the authenticated state, the client must then select a mailbox to access, and the server checks if the client has permissions to this box (by checking against the username). Both sides then move to the selection state.

In the *selection state*, a user has been identified and has selected a mailbox to review. Now, the user has information, maintenance, and downloading choices much like those with POP3. However, there are more possibilities for maintenance, because IMAP allows messages to stay on the server and to be placed in folders for storage. IMAP does not assume that the messages are going to remain on the receiver's device.

The termination phase is initiated by the client. The server updates its tables and severs its connection with the client.

IMAP is an improvement over POP3 in two ways. First, it allows the client to have choices over the mailboxes it opens, allowing multiple mailboxes for a single client. Second, it allows storage of messages on the server in an organized manner. Security is not improved, however, over POP3. Plain text passwords and messages are still used and can easily be snooped and stolen.

Experiment with IMAP4

You can experiment with IMAP4 as easily as you did with POP3. Identify a IMAP4 server. This could easily be the one you use to download email from. Follow these steps:

1. Telnet to this server on port 143. Use a command line something like this:

```
telnet pop.mailserver.com 143
```

2. You should get an identifying response from the mail server. You then identify yourself like this:

```
sessionID LOGIN jipping jipspassword
```

   The "sessionID" is a string to identify the IMAP4 session you are engaging in. Substitute your username for "jipping" and your password for "jipspassword".

3. Once you have moved into the authenticated state, you must select your mailbox. The INBOX will do:

```
A001 SELECT INBOX
```

4. If you are successful, you should get a list of mail attributes along with an "...OK...SELECT completed" response, and you are in the selection state. In this state, you can manipulate the mailbox you have selected by using selection commands. There are many of these, but you can play with some:

```
A001 FETCH <msg#> (RFC822)
```

This will fetch a message in RFC822 format—normal Internet email format—from the server. Another is

```
A001 SEARCH FLAGGED SINCE 1-Feb-2001 NOT FROM "Smith"
```

This will find the messages since 1 February 2001 that are not sent from "Smith".

5. When you are done using the IMAP4 session, you can quit the session by issuing the LOGOUT command:

```
A001 LOGOUT
```

5.1.2 SMS Messaging push model

Email messages are meant to be in a general form that adapts to most computer systems. They are text-based, so most devices can read and relay them. They are flexible enough to accommodate many different types of data objects. By contrast, SMS messages are very specific and targeted by their nature to a specific carrier technology.

SMS messages are short messages, 160 characters or less, and are specifically targeted for mobile phones and devices that use a GSM phone service. They are data messages, not intended to be viewed by humans until they are decoded and displayed. The sending of SMS messages adheres to the "push" model of message delivery. Messages are sent to a service center that relays the message and delivers it to its destination. The service center contacts the device—and keeps trying until it finds the device powered on and receiving messages.

An SMS message follows the standard message format in that there is a message header and a message body. The header contains information about the message and the body contains the message itself. The SMS message below contains the message "hello":

```
07917283010010F5040BC87238880900F10000993092516195800AE83229BFD46
```

Note that the data is actually a stream of bits and written above in hexadecimal. If this message were to be received on a GSM phone, chances are that the phone would display "hello". It is fairly obvious that we are dealing with a different type of message than we dealt with for email.

Let's examine this message and by this example examine the SMS standards for messaging. The table below analyzes the pieces of the example SMS message above.

Data Field	Description
07	The length of the service center information. In this case, the number is 7 octets.
91	The type of service center address. In this case, 91 means the phone number has an international format.
72 83 01 00 10 F5	The service center address in "decimal semi-octets". Although formatted like octets, the number reads in decimal digits. Because the service center address has an odd number of digits, it is padded with F (all ones) to pad out the octet. Here, the number is +27381000015.
04	Type of message. This is an SMS-DELIVER message.
0B	Sender address length. Here, 0B means the sender address is 11 digits.
C8	The type of the sender's address
72 38 88 09 00 F1	The sender's address, in decimal semi-octets. Note the padding. Here the address is +27838890001
00	A protocol identifier, establishing the way we will send the rest of the messages.
00	The data coding scheme. SMS messages can be sent in many encoding; the most popular, used here, is 7-bit data.

Data Field	Description
99 30 92 51 61 95 80	This is the sending time stamp in semi-octets. The first 6 octets represent the date, the next 6 represent the time, and the last two represent the time zone.
0A	Length of the message, in this case 5 octets.
E83229BFD46	The actual message, where 8-bit octets are used for 7-bit data.

Let's take note of a couple of things about this message. First, the semi-octet format of the addresses and the timestamp is odd, but readable. Note that the octets are swapped in this representation. Second, this message is in 7-bit "default alphabet" format. This is an alphabet of 127 characters that contains many of the most often used international characters. This is a GSM standard. Finally, to compress as much as possible, the 7-bit representation is encoded in the 8-bit quantities in a special way.

SMS messages can have many forms. In the GSM standard, the messages can, for example, be faxes or pages. The standard also allows email messages to be sent to GSM phones by way of SMS. In this latter case, the mobile device is able to treat the message as an email message and perform email operations on it (like replying to the message, for example). Adapting SMS messages to these other forms requires both a service provider that can perform such conversions and software on both the sender's and receiver's devices that can handle these adaptations. For example, the service provider will probably have to provide an email-address-to-phone-number mapping and software must be used to cut large messages into smaller messages that can be sent over SMS.

5.1.3 BIO Messaging

BIO messages are messages meant for the receiving device, not the user. These messages contain structured data objects of a known, predetermined format. They can be delivered using various transports: for example, email, SMS, and IR can be used.

Various data objects can be sent as BIO messages, including SMS configuration messages, configuration settings for various applications, and application data objects.

A good example of a BIO message is one that contains ringtones. A ringtone is a tune that a mobile phone will play to alert its user of some condition—say, a call coming in. While mobile phones have many unique tones, many phone manufacturers also provide the ability to program ringtones and send them to the phone. So you can have your favorite movie theme song play when you get a phone call. Ringtones are programmed using a textual "language" that can be encapsulated in an SMS message and sent to a mobile device. Because of special settings in the message header, the phone would process the message itself rather than display the message on its screen. Processing a ringtone means decoding the specification, storing the resulting tone, and incorporating the tone in its list of tones.

Another good example of BIO messaging is the exchange of vCards. vCards are electronic versions of business cards, containing names and contact information. The specification of a vCard is textual and can be included as an object, for instance, in email. By sending a device your vCard, by IR, for example, your contact information can be automatically inserted into the device's directory. This is accomplished by sending a message containing the textual specification of a vCard. By indicating in the message header that the vCard message is a special message, the device intercepts the vCard and records it, rather than displaying it.

vCard and vCalendar Objects

Virtual business cards and virtual calendar objects are a standardized way to exchange information about contacts and agenda items between devices. These can be attached to email or sent to another device via methods like IR or Bluetooth. They are textual specifications and, as such, are flexible and adaptable to many different transports and devices.

The Internet Mail Consortium governs the maintenance of the standards on these objects. The IMC is a group of computer companies that includes Symbian. Their Web site is at ***http://www.imc.org***.

Further information about the format of these objects can be found in later chapters. We will be using these objects in examples; they are supported directly by the Symbian platform.

5.1.4 Fax Messaging

As a messaging technology, facsimile transmission—that is, the electronic transmission of images over phone lines—developed

in parallel with computer communication. As standards were developed regarding digital messages and their transmission, fax standards were developed independently. In order to integrate a fax standard with the messaging standards we have discussed, some adaptation has been required.

A fax message is actually an image. Before computers were used to send faxes, fax machines were developed to scan a piece of paper into an image in the machine's memory, to transfer this image to other machines via a modem, and finally to print the image from memory back to paper. As computers got involved in this process, they eliminated the need for paper, and the fax image could be converted from its native format to one of the more standard image formats that computers use (e.g., GIF or JPEG formats). Modems have been adapted to include fax capability.

The faxing model is compatible with the messaging models we have discussed. Sending a fax follows the "push" model of messaging, where the sender keeps trying to send the fax until the intended recipient fax machine can receive it. As a message, a fax transmission has a body: it is the image that gets transmitted. We have to stretch the model a bit to find a message header, however. For a fax message, implementations typically consider the cover page of the fax transmission to be its header. It contains information typically found in a message header—send and destination information, for example—and it precedes the message body.

The data format of a fax transmission is specified by the Comité Consultatif International Téléphonique et Télégraphique (CCITT), now known by its parent organization, the International Telecommunication Union (ITU). The actual format of the graphics image that gets transferred is specified but is outside the scope of this book. It is the job of the sending machine to convert any text or images to that graphics format, to use the fax mode of an attached communications device, and to push the graphics data stream through to the recipient.

Sending a fax is typically done through a modem with fax capability. There is also faxing capability built into the GSM standard. Mobile devices that use GSM therefore have the capability of sending fax messages through the GSM service.

The Fax Image Format

You can find more information on the fax image format on the Web. However, CCITT Group 4 standards are a bit hard to find, because fax images are specified as TIFF class F images. More information on TIFF class F can be found in "The Spirit of TIFF

Class F", 1990, available from Cygnet Technologies, 2560 9th., Suite 220, Berkeley, CA USA.

5.2 World Wide Web Technologies

It is fair to say that when Tim Berners-Lee started to build a system of "distributed hypermedia" (as he called it) in 1992, no one suspected that the eventual result—the World Wide Web—would be as large or as important a system as it is today. As it has currently evolved, the Web is a gigantic collection of interconnected documents that use the ideas of hypertext and multimedia to link documents to each other and to convey information. The Web contains many different forms of content, mostly distributed via the Internet.

The Web is a collection of hypertext documents. These documents contain text and images, formatted according to certain rules. The hypertext nature of Web pages is what has made them such a useful tool; buried in the text and graphics of a Web page are hypertext "links" to other documents and multimedia data on the Web. Links can reference Web pages, sounds, documents of any format; any downloadable data can be linked in this way. In a hypertext document, if you want more information about a particular subject that contains a link to another document, you can click or tap on the link to load the new page or to access the referenced data.

Using the Web is a classic example of client-server interaction. To access documents on the Web, you run a Web browser application as a client. The browser requests and downloads documents from Web servers, which serve up pages and other information. Web browsers can, in addition, access files by other protocol methods, for example, file transfer protocol (FTP), Internet news services (via Network News Transfer Protocol (NNTP)), and email (via POP3 or IMAP4). On top of these, if the server has search capabilities, browsers can enable searches of files and databases.

Pages on the Web are addressed by their Uniform Resource Locator (URL) address. This address is in the form:

```
<protocol>://<host>[:<port number>]/<file location>
```

The standard values for "<protocol>" are "http" for Hypertext Transfer Protocol, "ftp" for File Transfer Protocol, and "mailto" for access to SMTP email servers. This set has been expanded by various browsers for certain uses (e.g., "about" and "file" are popular). The "<host>" field specifies the returned address of the computer (either name or IP address) and the "<file location>"

field specifies the path to the file on that host. The <port number> is optional and gives the TCP port number on the <host> server to use to initiate the connection and to send the request. Most Web service is done over port 80.

Web servers implement many different services, usually accessed through some kind of dynamically generated Web page interface. A common way to access the dynamic features of the Web is through the Common Gateway Interface (CGI). Via CGI, the Web server implements a gateway to its execution capabilities. Using CGI, a Web page reference will cause a separate program to be run on the server side and that program's output will be channeled through the Web server to the browser as the Web page. Dynamic Web pages are also implemented using other methods:

- Web pages can reference Java programs, which can be executed by a Java-enabled browser right in the Web page.
- JavaScript, a Java-based scripting language, can be included in a Web page and can be used to make the display of the page programmable.
- Active Server Pages (ASPs) are pages generated dynamically by the Web server, which uses scripts contained in the Web page to build the page that is sent to the browser. ASP files can contain HTML (including related client-side scripts) as well as calls to Windows components that perform a variety of tasks, such as connecting to a database or processing business logic. ASPs represent a streamlined alternative to CGI for Windows servers.

So Web browsing and Web service can access static pages and can work together to implement dynamic interfaces. On the Symbian platform, our focus is on the client side of the Web—browsing and accessing pages. Therefore, two important parts of using the Web that warrant closer attention are HTTP—the Web page transport protocol—and HTML—the language used to specify Web pages.

5.2.1 HTTP: Hypertext Transfer Protocol

HTTP is the protocol used to send pages from a Web server to a Web browser. HTTP is, in fact, two different protocols. It specifies the interaction of machines to transfer the data that makes up a Web page. It also specifies the format of the Web page data. The Web page itself is augmented in HTTP to give the data stream some extra meaning for the Web browser.

Transferring the File Data

HTTP was designed as a simple, text-command-based protocol that could transfer Web pages and adapt to many situations. In its current version (the Symbian platform supports HTTP version 1.1), HTTP is a very stable protocol that works over a variety of connection strategies in a number of different network configurations.

HTTP was designed to work within a TCP/IP network framework, using a client-server model. In the simplest case, this can be accomplished with a single connection between the browser client and the Web server.

More complicated situations develop when intermediaries are present in the client-server relationship. Three forms of intermediary are possible: proxy, gateway, and tunnel. A *proxy* is a forwarding agent that receives requests for a Web resource, rewrites all or part of the request, and relays the reformatted request to the server in the original request (i.e., the URL). A *gateway* is a receiving agent, acting as a layer above some other server(s) and, if necessary, translating the requests to the underlying server's protocol. A *tunnel* acts as a relay point between two connections without changing the requests or responses; tunnels are used when the communication needs to pass through an intermediary (such as a firewall) even when the intermediary cannot understand the contents of the messages.

Any party to the request chain which is not acting as a tunnel may employ an internal cache for handling requests. The effect of a cache is that the request/response chain is shortened if one of the participants along the chain has a cached response applicable to that request. This shortens the response time, especially where several intermediaries are involved.

The HTTP protocol itself is a stateless protocol. This means that one request/response pair occurs per connection, so that neither side needs to keep any information about the state of the connection. This is optimal if we want to make the connection fault tolerant, and flexible for many different situations. This runs over a TCP connection, so that reliability is assured.

HTTP consists mainly of a request command: GET. There are various forms of this request, but it basically looks like

```
GET <url> <protocol version>
```

The Uniform Resource Identifier (the <url>) is in a full pathname form, one where the <protocol> and <host> portions of the original URL are removed. The <protocol version> is the version of HTTP that is assumed. It specifies the name of the protocol and the version number, such as "HTTP/1.1".

The response to this request is either an error specification (including error number and error message) or a stream of data that constitutes a Web page.

Experimenting with HTTP

A great way to understand HTTP is to see how it works first hand. You can access many Web servers through "telnet" on port 80. So, pick your favorite Web server, and telnet to it as

```
telnet web.server.com 80
```

You should get a connection, but no response past that. Then type

```
GET / HTTP/1.0
```

and watch the results from the Web server. This looks for the default page at the root of the server's document tree (usually "index.html" or "index.htm"). The connection should close after the page is delivered.

The Format of the Web Page Data

If no error results from the request, Web page data is to be returned. This data takes the form of a MIME message. That is, it contains a message header and a message body; the header directs the browser what to do with the body. It is possible that the header is empty; this would be the default case where the body contained only HTML code to be directly displayed by the browser.

The MIME message header is a series of lines in "<key>: <value>" format. The "<key>" can come from a large set of specifiers, but the most useful is the "Content-Type" key. This can dictate the character of the data that is contained in the page body, which in turn can direct the browser on how to display it. For example, the type "text/plain" would be displayed verbatim in the browser's display window, without formatting. The typical type is "text/html", for a typical HTML-based Web page. But many different types are useful, such as "application/ms-word", which might start a Microsoft Word application and feed the page body to it, or "audio/mp3", which might cause an MP3 player to be started using the page data.

5.2.2 Using HTML

Hypertext Markup Language (HTML) is the language used to specify Web pages. That is, HTML is the language recognized by Web browsers, used for formatting and content information. It is a standardized language, with the standard controlled by the World Wide Web Consortium. Symbian OS implements the current version of the standard, which is version 4.0.

HTML is an attempt to specify the properties of a document in a comprehensive, textual form. A document has many properties, including a title, a description, formatting elements, and content. Each property item is specified by a series of *tags*, which are specifications surrounded by angle brackets—that is, the characters "<" and ">". A properly formatted HTML document has a head and a body, and has each property tag paired with the appropriate concluding tag. Therefore, a correct template for starting an HTML document is

```
<HTML>
<HEAD>
   ...header information...
</HEAD>
<BODY>
   ...body information...
</BODY>
</HTML>
```

This is a minimal specification; tags can have parameters specified inside them, which can alter the configuration of the item being described.

Header information includes the title of the document, any keywords that might describe the document, any description of scripts that might be used by the document body, and comments describing the document to human readers. The body information includes formatting information, specification of special HTML content (such as forms or dynamic interaction areas), and textual content itself, which typically forms the heart of the document.

It is important to note here that HTML is textual and, with a few exceptions, rather simple in its formatting specification. The tricks to processing HTML is the parsing of it and its translation to a device's display. These processes require a fairly large amount of processing. Providing a manual on the types of tags possible or even about the many ways Web documents can be specified

and used is obviously beyond the scope of this book. However, it should be clear that Web pages download easily and require a fair amount of manipulation and processing when they arrive at their client destination.

The HTML Standard

The World Wide Web Consortium (W3C) is generally considered to be the maintainer of the HTML standard. The W3C maintains many protocols and Web markup specification standards for document layout. They also have browsing tools as well as Web page verifiers. You can access their Web site at ***http://www.w3.org***.

HTML is a subset of a larger markup language called Standardized General Markup Language (SGML). A set of resources on SGML can be found on the W3C Web site at ***http://www.w3.org/MarkUp/SGML/***. HTML has extensively expanded into a structured data description language called Extensible Markup Language (XML). Information on XML can also be found on the W3C Website at ***http://www.w3.org/XML/***.

5.3 Wireless Access Protocol

HTML browsing over the Web has proven to be an extremely popular and flexible method of content delivery. As small devices and wireless communication have developed, it has been very tempting to adapt the model of Web page browsing to a mobile, phone-based platform. However, HTML requires some extensive processing to display, and that processing can take a heavy toll on resources. In addition, downloading the amount of information for HTML pages can tax memory and bandwidth.

Wireless Access Protocol (WAP) was developed to circumvent the problems of "heavyweight" content while retaining the benefits of client-server-based hypertext information. We have seen that WAP requires a new protocol stack, one that streamlines content delivery yet remains layered and secure. The top layer of the WAP stack implements content delivery: the Wireless Session protocol is designed to deliver Wireless Markup Language to the display device.

5.3.1 WSP: Wireless Session Protocol

The constraints of the WAP environment has imposed design restrictions on the entire WAP stack. These restrictions can be seen in the

WSP layer. This layer is meant to emulate the HTTP protocol in a restricted, efficient way.

WSP provides two services to the application layer: connectionless and connection-based services. Connectionless service is a thinner service, providing unreliable delivery with no requirement for acknowledgment from the receiver. In this mode, the WSP layer talks directly to the WDP layer, bypassing the WTP layer and making a short path to the bearer layer. This is a thin transport mode with low overhead. It is the most useful for the delivery of small amounts of data in situations that do not require feedback from the recipient.

Connection-based delivery uses the WTP layer and ensures reliable, secure delivery of data packets. Connection-based delivery demands acknowledgements from receivers and implements retransmissions if the packets are lost. This mode of WSP also adds capability negotiation, header caching, and longer-lived sessions. These last three features emulate features implemented between HTTP browsers and servers. This delivery method is used for the delivery of larger amounts of data or data that is error-sensitive.

Symbian OS supports both connectionless and connection-based sessions by its implementation of the WSP layer.

5.3.2 Using WML

Wireless Markup Language (WML) is a language used to specify formatting of WAP "pages" through a WAP microbrowser application.

WML is based on XML, a markup language superset of HTML used to describe data sets and their semantics. WML uses a metaphor based on decks and cards: instead of pages, WML uses cards, and cards may be collected into decks. WML includes special link directives that navigate quickly between decks.

Like HTML, WML specifies formatting and layout of the document presentation (in this case, the card presentation). WML uses tags to specify formatting directives as well as card and deck configuration. A single WML document can have multiple cards specified so that one download can result in several cards linked and displayed efficiently.

As an example of WML, consider the WML specification below:

```
<wml>
   <card id="index">
      <do type="accept" label="Card 2">
           <go href="#CardTwo"/>
       </do>
```

```
        <do type="options" label="WSJ">
             <go href="http://wap.wsj.com"/>
         </do>
        <p align="center">
           Welcome!
        </p>
        <p align="left" mode="nowrap" >
        <select>
           <option onpick="http://wap.wsj.com"> Wall Street </option>
           <option onpick="#CardTwo"> Card 2 </option>
        </select>
        </p>
    </card>

    <CARD NAME="CardTwo">
        <P>
        This is Card Two!
        </P>
    </CARD>
</WML>
```

This WML specification describes two cards, one linked to the other. The specification is minimal, especially when compared to HTML, yet effective. The first screen that this specification describes is shown in Figure 5.3.

Figure 5.3 Example WML code displayed.

There are several other aspects of WML that extend it beyond displaying information. First, WML includes WMLscript, a scripting language that can be embedded in WML specifications for dynamic card interactions. Second, WML includes a Wireless Telephony Application Interface (WTAI). This interface allows WML to access the telephony functions of the wireless device that it is running on. For instance, using a URL starting with "wtai:" can make a phone call if an option is selected. Finally, access to the WTAI extends WML's functionality beyond just call control; through WTAI, WML can access address book data and phonebook services.

5.4 Other Content Technologies

There are other content technologies that the Symbian platform supports. Among these are the following:

- *FTP:* File Transfer Protocol (FTP) was one of the first methods developed to transfer files between devices over the Internet. Developed to run over TCP/IP networks, FTP actually implements two protocols: (1) a command protocol that is used to direct (2) a data link protocol that exchanges file data. FTP is a simple, robust protocol that is still in wide use today. The Symbian platform supports FTP directly through an application engine in version 6.1.
- *TELNET:* This protocol is less a content provider and more a content access protocol. Telnet is a remote access protocol that allows devices to access TCP/IP service ports on remote devices. The typical use for telnet is to access port 23, which is a remote execution shell service, providing access to command shells on remote machines. Telnet is supported directly on the Symbian platform 6.1 through an application engine.

5.5 Some Perspective on Content

As we leave this chapter, we should stop for a moment to consider the importance of content and the technology each type of content represents.

Messaging is a relatively mature content technology. People have long understood the strategic importance of messaging and implemented many ways to distribute messages to colleagues, coworkers, and customers. Messaging started as fax machine transmissions and local area network email. It used to be that only email between 8:00 AM and 5:00 PM was important because that is when workers could access a computer system to read it. Messaging evolved as global communication became important and timing constraints grew. Now, workers realize that when it is 3:00 AM in the United States (and I am probably asleep), it is 4:00 PM in Korea and someone might need to get me an important message that might even wake me up.

Messaging methods have different impacts on message readers. Email is a less urgent medium. Because messages gather on a mail server before being collected and read, they are subject to the time schedule of the person that reads them. SMS and BIO messages are more urgent and, because of their immediate delivery

methods, can be used for time-sensitive and immediate information. Fax messaging lies between these other message media. If it can be delivered to a Symbian OS device, it can be an immediate message. However, many times, a fax transmission is sent to a desktop computer or fax machine and therefore is subject to the same constraints as email—messages are read whenever they are picked up from a delivery location.

Recent studies have shown that messaging systems are a crucial part of businesses today. It has been said that the typical computer user spends about two hours each day, or about 500 hours each year, using functionality provided by the messaging system. Fully 35% percent of critical business information used by computer users is stored in a messaging system, either in individual mailboxes, in folders or within a database. This points out how critical messaging has become.

Browsable content represents opportunities for people to obtain up-to-date information instantly. Both HTML and WML represent many ways for keeping people informed. Companies keep their sales staff up on the information they need; employees can reference company information instantly; anyone can access a weather forecast. Integrating browsable content with databases and dynamic information retrieval has powerful potential.

5.6 Summary

This chapter covered the kinds of communication content that Symbian OS supports. Specifically, we covered three content types: messaging in all its forms, HTML content, and WML content. We covered the four kinds of messages Symbian OS currently supports: email, SMS, fax, and BIO messages.

The next chapter will conclude our look at the basics of the Symbian OS communication architecture by reviewing issues that are connected with data security.

6

Security and Communication

Eavesdropping is a very easy thing to do. It takes very little effort to listen to someone else's conversation, especially in an environment where you might go unnoticed. This is especially hard to resist if you hear your name used in a crowded room. In a social gathering, it is excruciating to not listen to a conversation if you think people are talking about you. It is much easier to walk close to the group, to fake some activity (as if you should be there), and to try to gather as much of the conversation as you can.

In the realm of data communication, eavesdropping is also an easy thing to do. As we reviewed in Chapter 3, when data moves between two devices, an opportunity exists for snooping on that data. In some cases, it is even possible to tamper with the data or to confuse the receiver. Like voices in a room, new voices can be introduced that confuse listeners or drown out speakers.

In this chapter, we review issues that surround the security of communication and the solutions supported by the Symbian platform. We will review ways of snooping on data; we will overview the ways data can be protected; and we will highlight the methods implemented by the Symbian platform.

6.1 An Overview of Security Issues

In general, security is all about *control*. Obtaining and retaining control over both data and access are the keys to security. Naturally, this control is easier to manage when the data remains on a single computer, or when there is no access possible. But when the data leaves a device, or the device is connected to a network, control becomes difficult to manage. There are two areas where we are concerned about data control: *physical issues*, where outside parties can use the physical properties of communication (e.g., cables or

light) to eavesdrop on data exchanges and perhaps manipulate those exchanges, and *data issues*, where outside parties can exploit or change the data itself. Access control is also of concern here. In addition to data control, controlling access to devices—both physical and electronic access—is very important, even to small devices.

Along with issues of control, *trust* is also foundational. If we cannot control data completely, then we must be able to trust the entities that can see our data. To avoid gullibility, trust must be granted, not assumed. We grant trust to entities that can prove who they are. So we must be concerned with trust issues like authentication and identification, as well as issues of granting trust.

6.1.1 Physical Issues

The eavesdropping, or snooping, that can be done on data is a major problem in communication. In the brief milliseconds that it takes to transfer data packets from one device to another, many other devices can possibly view and decode those packets. While not all types of transport are susceptible, many transport methods are very open to data snooping.

In general, wired serial communication is extremely difficult to snoop. There is no way to "listen" to a serial cable, unless you splice into that cable. This can indeed happen, especially when cables go for long distances that are not monitored (for example, between buildings or traversing floors). However, without physically altering the environment, eavesdropping on a serial data cable is not feasible.

Wireless serial communication, however, is much easier to snoop. With infrared communication, the light beams cover an area approximately 15 degrees on either side of center from the origin. So if the intended receiver is on one side of the field, a snooping device could be on the other side of the field and still be able to receive a signal. If the sender receives protocol acknowledgements from multiple receivers, it will get confused and stop sending. But a "collector" that simply recorded an IR session and later disassembled it to figure out the data exchange is certainly possible. However, you would think that, with the short range of IR and the need to point directly at the sending device, such a collector would be noticed very quickly.

Network technologies are notoriously easy to snoop. As an example, consider Ethernet. Ethernet is a passive medium, broadcasting its packets across a network. It depends on machines behaving correctly, that only the machine that is the intended destination of a packet will pick up and decode that packet. This

is a matter of trust, of course, but this trust is *assumed* to exist. However, nothing stops any connected machine from listening to all packets, collecting all data, and reassembling a conversation between machines. The snooping device simply needs to be wired to the same network and needs to access the network in such a way that all packets are read, not just the ones that are meant for the listening network card. This is a special listening mode, called "promiscuous mode".

Wireless media are even easier to snoop. In a wireless environment, the snooping device does not even need to be wired into the network; it simply needs to be within radio range and have the right receiving equipment. Wireless snooping is easy because it does not look conspicuous. A data collector simply needs to be inconspicuously using his data device, and listening to data traffic at the same time. Wireless Ethernet and Bluetooth can be snooped this way.

Try This Out for Yourself

There are several ways to snoop communication. Try these methods to "listen" to an HTTP client-server conversation browsing Web pages over port 80 on a TCP/IP network.

If you are using a Unix system, you might try the "tcpdump" command:

```
tcpdump tcp port 80
```

then access a Web page through a browser. This command puts the network card of the machine you are using into promiscuous mode and reads all network packets. It reports TCP packets that are sent and received on port 80, the default port used for Web page service. The output can tend to be cryptic, but much information can be assembled, including sites visited and actual HTML of the Web pages browsed.

If you using a Unix workstation that runs Solaris (from Sun Microsystems), the "snoop" command will help you more. Use the command

```
snoop -V port 80
```

This will print the conversation across port 80 in a more readable form.

Note that in Unix systems, putting the network card in promiscuous mode and listening to all packets requires special administrator privileges.

There are not many free software packages for Microsoft Windows installations that will sniff a network. Most software packages are protocol analyzers and cost quite a bit of money.

For Symbian platform devices, there is a package called "NetSpy" that will sniff network packets. It is written in a combination of Java and C++, and allows users to write "plug-ins" into the application that will analyze network traffic.

Most snooping on communication is not preventable. Because any party can connect and listen to most media, and most of the time can do so undetected, security is typically not concentrated on physical media.

6.1.2 Data Issues

Physical issues deal with access to data; data issues deal with what to do with the data once you have it. Data issues focus on *understanding* data, *using* data, and then *manipulating* data.

Understanding the data obtained through snooping can be harder than you think. Let's say that you have successfully snooped a Web browsing session that includes someone accessing email through a Web server. What you actually have is a large amount of packet data—a direct recording of data frames wrapped in data frames as application layer data moves down a protocol stack to eventually go out over a physical medium. The prize data item you are looking for—say a username and password for the email service that was used—does not stand out for you to read. You must sort through the packet data to find it.

So there is a large sorting job in finding the right data. There are many applications that will help with this. These applications identify the protocol header of each layer in the data frames and can strip away that header to present the real data. Even at this point, however, getting the data you want is not straightforward. You still have to sort through a conversation between computers that may include protocol acknowledgements, raw HTML Web page data, error responses, and the like. There are also applications that will help with this kind of reconstruction, although there are not as many as there are protocol analyzers. This is especially true in younger transport media, like Bluetooth.

At this point, because the conversation between a Web browser and a Web server is not complicated, finding the username and

password you seek can be done fairly easily once you understand the way HTTP works. However, other protocol conversations are much more difficult, especially if you do not know the applications involved in the conversations. Again, applications exist that will help you with this, or at least can be written for specific types of data exchanges.

There are two points here: that sifting through protocol exchanges to obtain data is very difficult, and that it is not impossible. On a computer, when a job is "very difficult", it just means that a program needs to be written to do that job, and that the program might take a while to write. When that program is written, however, the job is no longer very difficult. With computers, the fact that something is "not impossible" simply means "it can be done".

Using data once that data has been understood is certainly easy for the snooper and dangerous for the owner of that data. Once you have obtained a username and password from a Web email session, it is easy to engage the same Web service and use the newly acquired information.

The possibility that information could be manipulated presents an even more dangerous scenario for the data owner. Here, data is changed somehow and inserted back into the data stream to be reused. It is possible to affect the original data or to affect the outcome of using the data. Consider a situation where a device was downloading data while moving in a car. As that device roamed between service locations in its service area, its data stream could be interrupted and fed from different points. An intelligent attacker could feed a different data set from different service points, thereby giving the receiver incorrect data.

Protecting against data snooping and manipulation takes some innovation. As we have seen, it is almost impossible to stop snooping from occurring. So, instead, we can scramble the data so that it is unusable to anyone but the intended receiver. *Encryption techniques* encode data using a special key in such a way that only those possessing that key can decode the data and use it. And, since the key now becomes an important data item, we can select the key in ways that only the sender and receiver understand. To prevent manipulation, we can also use methods of verification. The most effective verification methods involve using credentials—special identifying information—that a sender has received from a server, ones that the receiver is expecting (from contacting the same server). Presenting credentials before sending data and forcing the receiver to verify those credentials makes it very tough to receive data from an unexpected source.

Certificates

Certificates are used by the security implementations as a way to identify remote servers and to obtain their encryption keys. A certificate is a digital document that "vouches for" a computer system. They are typically issued by third party companies 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 must be verified when they are used. This is done by contacting the issuer, obtaining some certificate information, and comparing it to the certificate presented by the remote device.

6.1.3 Remote Access Control

There are several other ways to lose control of a computing device. Almost by definition, establishing any kind of service gives control away. Servers that open up the serving device to remote access can potentially jeopardize its security.

It is the nature of servers to create this potential problem. Servers expect and invite connections from sources both inside and outside their computing environment. They are usually passive in accepting connections and do little to verify the sender. Once a connection is made, a server will typically accept any data that is sent and process that data immediately to provide a response. By definition, servers do this as quickly and efficiently as possible.

The problem that creates security holes is the level of trust that must be established between the server and the client. There is a certain balancing act that happens here. If the server is going to be fast and efficient, then it will try to trust the client more. If the server does not trust the client, then service will be slower as the server employs encryption, verifies credentials, or examines data before responding to the client. Most servers either ignore trust issues altogether, because their service is too simple to worry about trust issues, or try to strike a compromise between trust and efficiency by assuming a kind of middle level of trust that can be verified easily.

There are more servers on a computing device than you might suspect at first glance. Any time a device is open to receive data, there is a kind of server running that will accept and respond to that data. For example, consider the following situations:

- A *Web server* is always waiting for requests to come in, and collects any data traffic over the port it is listening to. If that data does

not make sense, it responds with an error message. However, note that any data can be sent through the listening port.

- A *WAP browser* is open to data between the time it requests pages from a server and the time those pages arrive. Between these two points, it will accept and analyze data.
- As two devices exchange data *using infrared*, each side accepts a data stream for a certain length of time. As we saw in Chapter 4, each side transmits for 500 ms at a time, then stops and listens for data from the other side.
- A *Bluetooth device* in discovery mode makes broadcast requests and waits for responses.
- Any *user program* can use the data ports on a device to send and receive information.

In each of these situations, there was an application of some sort waiting to receive data of a certain form. Notice the level of trust across these situations can vary widely. A Web server will typically accept any request from any source but only process requests that make sense. A WAP browser is conversing with a server through a built-in layer that encrypts the conversation by default. The user program that is sending and receiving data has probably ignored all security issues to concentrate on getting the data exchange right, assuming that security is a feature to be added later.

We can prevent remote access problems by not making assumptions. When we assume that data will be in the right form or that a request cannot come from a malicious source, we expose our naiveté and invite security problems. By assuming that all data is not correct or that all connections might not be error-free, we will increase processing time but are wiser in our approach to outside connections. The penalty here is not so severe; the Symbian platform has several security measures built into it that are efficient and fast to use.

6.1.4 Threats to Symbian OS Devices

We just outlined three areas of security vulnerability: physical issues, data issues, and access issues. Let's consider how these issues impact Symbian OS devices.

Physical issues are important, but device users can do little about them. As we stated previously, snooping on all the various types of media is extremely easy. Mobile telephony signals and wired Ethernet traffic is easy to listen to if you have the right equipment. Since we have no choice as to the medium that is used to transmit data, physical issues are out of most device users' hands.

Data issues are extremely important. Because snooping is so easy, we must assume that any data exchanged between two devices is vulnerable. Considering the kind of data that might be exchanged—anything from personal to electronic commerce information—data encryption is necessary for any transaction.

Because data issues are so important, access issues are important as well. It is unlikely that an attacker will "break into" a Symbian OS device. However, control over which external devices can request information from a local device is important. Control over the exposure of data from your local device is also extremely important. For example, the services offered from a Bluetooth device can expose it to probes and access from other devices without warning.

6.2 Security Measures

6.2.1 Authentication

One way that is often used as verification of identity—most often for remote access—is authentication. Authentication is what happens when a computer or user identifies itself and another computer verifies the identification. Although it is not completely secure, this means of security has the longest history and is still widely used.

There are two types of information needed from the initiator of the authentication process:

- *Identification:* The initiator—computer or human, but most often human—must provide some form of identification. This can be in the form of a username, a real name or even an identification number. The key here is that this piece of information uniquely identifies the initiator.
- *Private information:* This is information known only to the initiator and the computer verifying the authentication attempt. This is typically a password, but it can be one of a number of things. More recently, items like challenge-response information (where the computer prints a random number, and the initiator responds to that number in a unique way) or biometric data (some measure based on the pattern of your fingerprint or blood vessel organization in your retina) have been used.

The idea is that together these data items can uniquely identify the initiator of the authentication. Once these information items are entered correctly, the initiator gains access to the required resource. Typically, authentication is used to gain access to remote shell-based or network services.

There are two large holes when we use authentication as a security scheme. They are (1) that the identification information cannot be verified and (2) that the private information can usually be derived or deduced. The basic problem is that authentication is designed to be used by humans. People cannot easily memorize long strings of random letters for user names or strings of random numbers for passwords. This means that identifying information is usually mnemonic, like a last name ("jipping", for instance) or some mutation of a name of the user ("thejipster", for example). Programs written using rules to guess identification can guess many forms of identifying information. In addition, private information is also easily guessed by using rules that people use to remember such things. A spouse's name or a boyfriend's birthdate are easy to remember but make very poor passwords.

Authentication schemes are used in many places. POP3 and IMAP4 mailboxes are protected with authentication systems; most entry via remote access to execution shell services is "guarded" by authentication systems. Use of PPP to access networks is based on authentication. Typically, the systems "protected" by authentication are legacy systems; newer access methods usually employ encryption or some kind of challenge-response system.

Choosing the Right Password

Choosing the right password is essential for good authentication protection. Unfortunately, the process runs counter to the schemes people use to remember such things. You should not choose passwords that could be feasibly listed in a dictionary. Permutations such as spelling a word or account name backwards, concatenating one or more words together, and prefixing or suffixing dictionary words with letters or digits are also bad choices for passwords. These are among the first permutations a dedicated attacker will check.

Characteristics of good passwords include sufficient length, complexity and obscurity. Use passwords that are at least 8 characters in length, contain random (as near to random as you can remember) combinations of letters and digits, use both upper and lowercase characters, and do not designate something that is real or can be guessed. Choosing a password that can be typed quickly, so somebody cannot follow what you type by looking over your shoulder, is an asset.

Assess yourself on how you have chosen passwords for your accounts. Do you use easily remembered (or guessed) passwords? Do you use the same password on several different accounts?

Ask your friends or coworkers how they have chosen passwords. How would you assess your authentication security? Do you change your password often?

When one device masquerades as another device, we call this *spoofing*. For example, firewalls on the Internet can be set up to receive data from specific IP addresses. People use this to restrict the computers that can access their data. It is possible for some computers to change their IP addresses to match those the firewalls will allow through, thereby seeing data they would otherwise not have access to. Or again, when you are engaging another person in an instant message session between Bluetooth devices, it is possible for another device to take over the session by masquerading as one of the conversing devices. (Note that I used the word "possible". It can be quite difficult to choose the device identification to accurately spoof another machine, especially when using Bluetooth.)

6.2.2 Security Attacks and Denial of Service

There are different threats to security other than those based on data. These threats typically combine some of the methods discussed above to actively attack computing systems or devices, or to deny some kind of service to devices. While these threats have been designed for larger computer systems in the past, they are being retooled for small devices and can be very insidious.

A device *attacks* another device when it tries to break through its security protection. Because the attacks are typically programmed and are done from one computer to another, they can go on endlessly until security is breached or the user that started the attack gets bored and stops it from happening. There are several types of attacks relevant to small devices:

- *Attacks on authentication:* Often called a "dictionary attack" or "password attack", these assaults make repeated attempts to break through authentication barriers by guessing the identification or private information, interpreting the responses, and trying again with a new guess.
- *Attacks on services:* These types of attacks target known bugs in the implementations of services. The idea is to either crash the service or to put the implementation into some kind of error mode that gives access to other system functions (like shell access). This is usually done by accessing "boundary conditions": overloading internal buffers or using untested commands.

On a small device, a crash can be lethal, bringing the device down to an inoperable state and potentially losing valuable data.

- *Attacks on protocols:* Again, these attacks focus on bugs in protocol implementations. The idea here is to force the device into a state that will accept (and execute) any command or simply freeze the machine.

One dangerous type of attack stands out from the above list: the *denial of service attack* (DoS). A DoS attack is not an attack on security; it does not try to break in to a system. Rather, a DoS attack tries to keep a device so busy that it cannot service requests from or make requests to other devices. There are several ways to keep a processor busy. Logins from nonsensical users that require lookups and responses, repeated rapid queries from networks that require responses and constant requests for the same Web page that require the same page sent over and over again are all good examples of DoS attacks. Not only is the processor on the attacked device kept busy, but the network pathway used to access that processor is kept full of these nonsensical queries, allowing no room for other data packets.

Distributed denial of service attacks (DDoS) are even more malicious. For DDoS attacks, multiple devices throughout a network stage an attack on a device. Usually, this distributed attack is orchestrated by a single attacker that channels network traffic through distributed points over a network, often without them knowing. DDoS attacks have been used over the Internet with a large degree of "success" (if you can call it that); DDoS attacks have brought down service to the Web sites of large companies by literally using thousands of computers to make millions of Web page queries, thereby shutting out actual users with real data requests.

Dissecting a Smurf

A Smurf attack is DDoS attack that combines a knowledge of Internet protocols with spoofing. At the IP layer, a "ping" is a query packet (called an ICMP packet) that asks some simple information of the receiver—normally as simple as "are you there". As a response, the receiver replies to the sender identified in the IP headers. A Smurf attacks puts a different address in the sender field than the real sender, then sends off the ICMP packet.

The result is that the receiver replies to an address that is *not* the sender's address. By using this technique, an attacker can send thousands of packets to the same final destination through many different sources, and the data storm that results looks like

it is coming from all over the network! The actual source of the attack is difficult, if not impossible, to track.

The official name for this attack is a "forged ICMP attack". Further details can be found from the Computer Emergency Response Team's Web site: ***http://www.cert.org/advisories/CA-1998-01.html.***

There is very little a computer can do by itself to thwart DoS attacks. The best way to stop these attacks is to stop the attack at the source. Think of the DoS attacks as water through a hose. If you are getting squirted in the face with a steady stream of water from a hose, holding your hands in front of your face will succeed in keeping your face drier, but the rest of your body will still get wet. Finding the spigot and turning off the faucet is the best way to stop the flow. However, finding the source of a DoS attack can be very difficult, especially when that attack is from a distributed set of sources. Often the best you can do is to clamp off the hose at the squirting end—to cut off the data flow as it enters your network. This often requires you to work with your communication service provider to cut off the data flowing into your network.

Note that for a mobile Symbian OS device, DoS and DDoS attacks become less of a threat. The wireless nature of a mobile device means that attackers need to be close to the device to snoop its communication data and discover information they can use to attack it. Also, since such a device is unlikely to be a server, attacking it with a torrent of service requests will do no good. Consider, for example, a phone handset using GSM to download WAP pages. An attacker must be in the vicinity of the GSM conversation to snoop it and must use GSM if it is to jam the device with too much data. About the only thing an attacker could do is call the device many times, which would be quite expensive.

6.3 Securing Transport Technologies

Now that we know the pitfalls, what can be done to secure communication, especially the technologies supported by the Symbian platform? Several steps can be taken at the transport level.

First, we must concentrate efforts where the payoff is greatest. This means *not* working to secure physical transmission. While some level of monitoring will be able to tell if snooping is being done on some level, it is generally not worth the effort spent, because snooping cannot be prevented. As we have stated, the

nature of our communication media makes snooping possible and makes preventing it nearly impossible.

Therefore, we should focus on protecting data. Most importantly, we should use encryption wherever possible. For transport technologies, the only protocol stack that has this built-in is the WAP stack. The WTLS layer provides secure encryption and decryption as it moves data between the WTP and WDP layers (see Figure 4.5).

There is little security built into the rest of the transport technologies. There is very little to be done here, since these are standards set up by consortia and committees. One thing we can do is to secure authentication. Authentication is used in several technologies (e.g., PPP) and choosing a good password will help a lot in securing these.

6.4 Securing Content Technologies

Content technologies are implementations more under a user's control, but implementations of standard protocols—such as HTTP or SMTP—are designed by standards committees and computer companies and are outside a user's influence. There is little in the way of security protection in the standard content protocols.

One implementation that stands out as an example of this is Secure HTTP (SHTTP), which is used for certain Web pages with special Web servers. These pages are referenced by a URL with a prefix of `shttp:`. SHTTP passes all pages through an encryption layer before sending the page's requests to or reading the result from the destination Web server. In addition, the Web server must also implement SHTTP to work with the encryption process used.

SHTTP is a protocol specifically designed for HTTP transport. More generic solutions would help *all* transport and application protocols. A new layer has been proposed for the TCP/IP stack that would implement a generic encryption strategy for transport. The Transport Layer Security (TLS) layer has been proposed to go between TCP and IP in the TCP/IP stack. This would protect all transport data without the need for special protocols. However, TLS is not an adopted standard for TCP/IP. It has been implemented by third parties for the Symbian platform, but does not come as a built-in Symbian platform implementation.

Another example is a system called Pretty Good Privacy (PGP). PGP allows email data to be encrypted before it is sent and allows the decryption of PGP data when it arrives on a device. Because the standard TCP/IP stack provides no encryption in its layers, PGP is positioned as an extra application step between TCP and a POP3 email client.

The Symbian platform does include implementations of encryption standards, built into library interfaces for user applications. As you design your applications, and want to protect application data, you must remember that you need to implement your own methods of protection. We will cover how to apply the Symbian implementations for data in Chapter 10.

6.5 Summary

In this chapter, we have reviewed issues that surround the security of data communication between devices. We reviewed issues about physically snooping on communication media and data issues about dealing with the data picked up when we snoop. We also examined issues that surround remote access to devices. We looked at authentication issues and did a brief overview of spoofing and service attack methods. We wrapped up the chapter by discussing ways to secure both transport and content media.

This chapter concludes the first section of this book, dedicated to examining the basics of communication on Symbian devices. The next chapter, Chapter 7, kicks off the section that serves as the meat of this book—how to write applications that use the communication infrastructure of the Symbian platform.

Section 2

Programming using Symbian OS Communication

7.	Getting Started with Communications Programming	pages 123–139
8.	Serial Communications	pages 141–169
9.	Communicating via Infrared: Serial Communications	pages 171–176
10.	Using Sockets	pages 177–194
11.	Communicating via Infrared: Using Sockets	pages 195–210
12.	Communicating with TCP/IP	pages 211–234
13.	Bluetooth Communications	pages 235–274
14.	Telephony	pages 275–297
15.	Sending and Receiving Messages	pages 299–332
16.	Browsable Content Technologies	pages 333–346

7

Getting Started with Communications Programming

This chapter introduces the section that is at the heart of this book: communications software development for Symbian OS. We have covered a lot of conceptual ground to this point. Now it is time to get to work and get your hands dirty.

To develop software you need a set of tools to write, compile, and debug programs—an "environment" for writing software. The software development environment that is used to develop for Symbian OS is a collection of tools: a combination of compilers, libraries, code generators, and emulators from several vendors. This chapter will overview what you need, where to get it, and how to use it. We will start by looking at the software development process and each tool in the Symbian software development toolbox. Then we will focus on the emulators, and how they can be used. We will finish the chapter by looking at how the development environments relate to real hardware—how the emulators use peripherals to target real devices. As we overview the toolsets we need, remember that my intention is not to give a full tutorial in using the Symbian OS development environment; it is, rather, to give an overview with some attention to certain areas that we will be using later.

As we get started, we should establish what environments are available. Symbian releases software development kits (SDKs) in various forms targeted at specific reference designs. Starting with EPOC Release 5, SDKs are distributed free of charge from Symbian's software developer support Web site. As of the writing of this book, the following SDKs are available:

- *EPOC Release 5:* You can get SDKs to develop for EPOC Release 5(ER5). These kits are released for the C++, Java, and OPL languages.

- *EPOC Release 5 Connectivity:* An SDK is also available for those who wish to develop software that uses PLP to communicate with a PC.
- *Symbian OS v6.1 for the Quartz reference design:* The current SDK is for version 6.1 of Symbian OS, using the C++ and Java languages. This SDK is available on the Symbian DevNet site for the Quartz reference design (see below).
- *Symbian OS v6.1 Connectivity:* This SDK assists developers writing software to integrate Symbian devices with PCs via PLP.

Note the representation of reference designs in the SDKs above. EPOC Release 5 did not make distinctions for various reference designs, and therefore there is only one release of the SDK. For the current Symbian platform, only Quartz reference design releases of the SDK are available through Symbian DevNet. SDKs for other reference designs are available from licensees that are manufacturing them. At the time of writing, Nokia has produced two reference design SDKs: one for a Crystal reference design (embodied by the 9210 communicator) and one for its Series 60 reference design (as seen in the 7650 phone).

Where To Get SDKs

The Web site for this book has links to the sites on the Internet where you can obtain the SDKs. You should check out the Symbian DevNet site for SDK patches and updates. In addition, licensees such as Nokia (see ***http://forum.nokia.com*** for SDKs for Crystal and Series 60 reference designs) have SDKs for their products.

Since the only SDKs available from Symbian are for Quartz, the software examples in this book will be given for Quartz using the SDK available from the Symbian DevNet site. However, since other SDKs we have mentioned are based on Symbian OS version 6.*x*, most of the functionality will translate to the other SDKs you can get. (N.B. Some will not: Bluetooth support was not complete in the version 6.0 SDK, for example.)

7.1 The Symbian Software Development Environment

Software for Symbian OS is developed using a combination of Symbian tools and libraries, Microsoft development tools, and compilers from the Free Software Foundation. There are essentially

two targets for software development: the emulated machines and the real hardware. The Microsoft development tools provide an environment for editing, compiling, and debugging for the emulated devices through its Visual C++ product. Compilation for real hardware is done through the GNU C++ compiler from the Free Software Foundation. Symbian tools and libraries work for both these environments.

The software development process starts with a problem that needs to be solved. Good software design attacks this problem by specifying a solution with good old-fashioned pencil and paper. A written problem statement combined with a written specification of a software design is the best way to end up with a good application. This written design is then used to write algorithms (in some pseudocode language, perhaps English) that will implement the solution statement.

Once all of this design has taken place, a computer program can be developed to implement the algorithms. Figure 7.1 encapsulates the development pathways into a small diagram.

Development begins with the programmer expressing the algorithms in C++. C++ works best when its object-oriented properties are emphasized (over its C origins) and an application's source is expressed as a collection of classes or modules. Each class has a specification and an implementation, usually given in `.h` and `.cpp` files. So before we start putting the application together, we can have a large number of source code files.

While the source code is the heart of an application, there are also many specification files involved along the way to a final, released application. They fall into some rough categories:

- *Project files:* Because there can be so many files involved in the construction process, project files are needed to direct the construction. Project files begin with an MMP file: the "makmake" project file. The `makmake` command takes an MMP file and turns it into a project file for the system that you will be using to construct your application. This might be a Visual C++ project file (a `.dsw` file) or a "makefile" for a build process using either the Visual C++ (`cl.exe`) or GNU (`gcc`) compilers. In version 6.1 of the Quartz SDK, for example, `makmake` can generate project files for 22 different platforms. Even with this variety, the most common are project files for Visual C++ and makefiles for the GNU compiler to construct ARM binaries.
- *Resource files:* To make applications more modular and adaptable, the Symbian SDKs support the use of resource files. These files contain objects—like test strings and GUI widgets—that

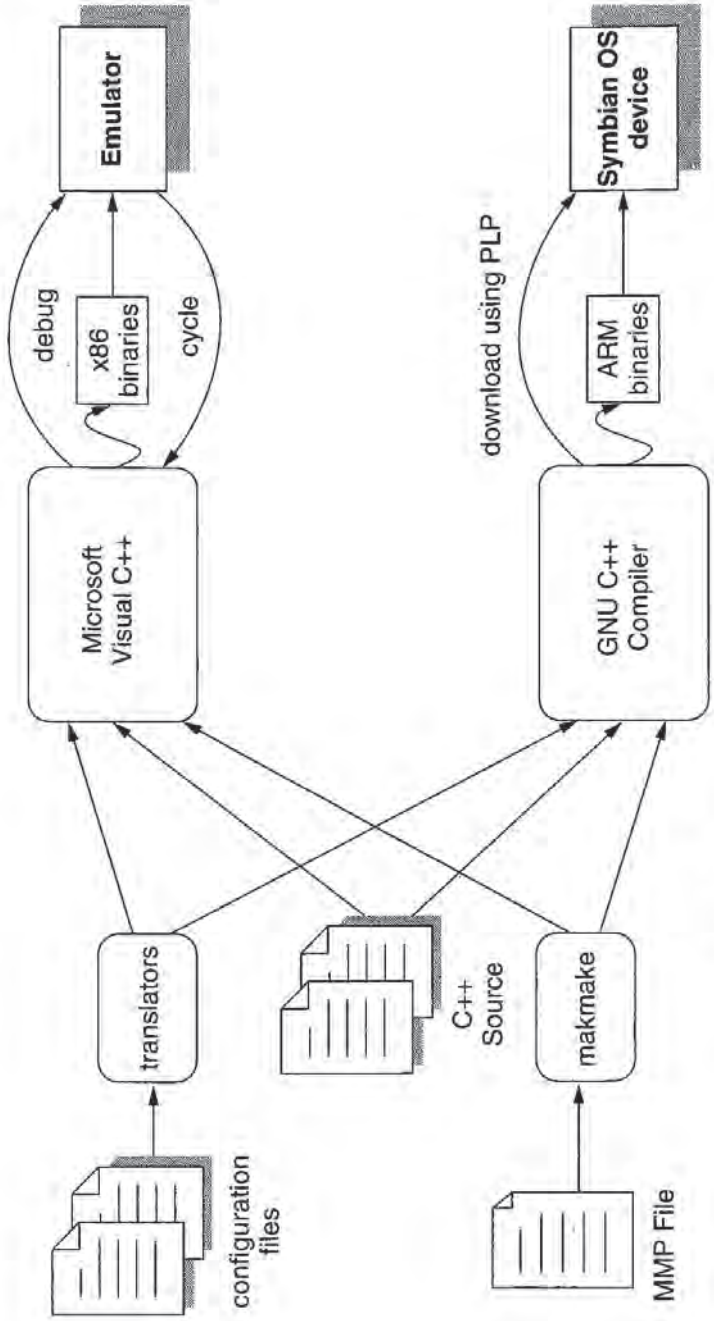

Figure 7.1 The Symbian software development path.

can change while the application's core remains constant. For example, strings used as an application's title could be represented in different languages and contained in separate resource files to localize an application for different countries. This would allow different resource files to accompany one central application executable to customize it. Menus, toolbars, buttons, hot key shortcut combinations are all examples of the type of item that would naturally be placed in a resource file. Resource files (using the `.rss` suffix) are translated into files used by the compilation process (such as the `.rsg` files that contain definitions for use in C++ code).

- *Help system files:* Help files used by applications that run under older versions of EPOC have been rather difficult to construct. They are in a database format, which has been specially designed for the help system. In version 6.1 of the SDKs, a new system is available to construct help files. Written in Java, it is a tool where help database files can be constructed, then attached to applications. If an application uses these help files, they must accompany the application for installation.
- *Package construction files:* The final deliverable product will likely be composed of several files—the application itself and libraries or configuration files. To enable easy installation of the files that an application requires, a tool called `makesis` is used to package them into a SIS file. `makesis` needs to be told how to create this installation package file, and a `.pkg` configuration file must be created to enable `makesis` to construct the package. In version 6.1 of the Symbian SDKs, a new tool called `sisar` presents a GUI to allow interactive construction of a package file.

At the end of this path is a set of executable binaries that can run on one of many platforms. As an example, the x86 versions that are built for the emulator are constructed either in the Visual C++ IDE or at the command line via the Visual C++ `nmake` command. As another example, the ARM binaries for target machines use the `nmake` command to drive the GNU compiler.

7.2 Hello World! A Simple Example

Let's illustrate this process with a simple example. It would be wrong to start off a book on programming without a "Hello World!" example, and the Quartz 6.1 SDK distribution contains several. Let's examine one and see how it is put together.

After installing the Quartz SDK, you will find a collection of files under the directory `Quartz\Epoc32Ex\quartzui\HelloWorld`. When they are all put together and the application is running, the emulator will display the simple message shown in Figure 7.2.

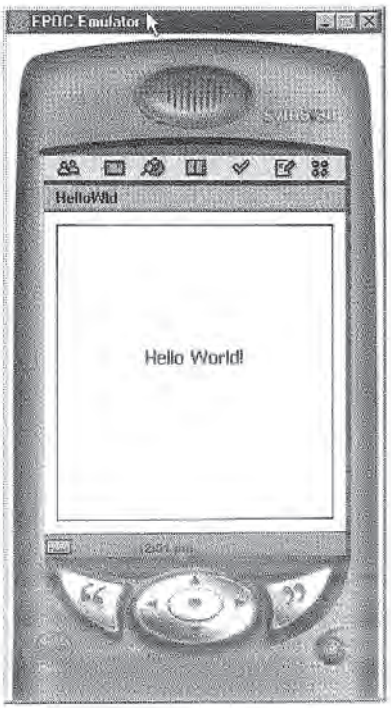

Figure 7.2 The Hello World! example.

7.2.1 The C++ Source Code

To start with, this directory contains source code for the `helloworld` example. There are four C++ modules and one `.h` file that specifies the interface specification for all of the C++ classes.

7.2.2 The MMP Project File

The MMP project file is called `helloworld.mmp`. The contents of this file are shown below.

```
// HelloWorld.mmp
//
// Copyright (c) 2000 Symbian Ltd. All rights reserved.
//

// using relative paths for sourcepath and user includes
TARGET        HelloWorld.app
```

```
TARGETTYPE     app
UID            0x100039CE 0x10008ACE
TARGETPATH     \system\apps\HelloWorld
SOURCEPATH     .
SOURCE         HelloWorld_Main.cpp
SOURCE         HelloWorld_Application.cpp
SOURCE         HelloWorld_Document.cpp
SOURCE         HelloWorld_AppUi.cpp
SOURCE         HelloWorld_AppView.cpp
USERINCLUDE    .
SYSTEMINCLUDE  \epoc32\include
RESOURCE       HelloWorld.rss
LIBRARY        euser.lib apparc.lib cone.lib eikcore.lib
```

The information in this file is given in key/value pairs, and is enough to generate the project files for the compilation systems we need.

- The end result is given by the `TARGET` and `TARGETTYPE` parameters. We are constructing an application (an `app` target type) and its name will be `HelloWorld.app`. 16 different target types are supported, the most common being for applications, DLLs, and console-style executables.
- The UID for this application is given as `0x10008ace` in hexadecimal (268470990 in decimal; both representations work). A UID is a 32-bit number that uniquely identifies an application to Symbian OS. Three identifiers are needed: the first to identify the *type* of the application (DLL, executable, or file store; in our example, `0x100039ce` represents an executable application), the second to identify the application itself (this identifier must be unique), and the third to identify components that work together within a single application. In this case, we have only one executable component and therefore can ignore the third UID.
- The paths for the target compiled applications and the source code are given by the `TARGETPATH` and `SOURCEPATH` specifiers, respectively. (Note for programmers who have used ER5: the source path can now be anywhere in the file system. In ER5 and earlier SDKs, the source path had to be exactly two levels deep.)
- The source C++ files are listed with the `SOURCE` tags. There are multiple source files, given with multiple tags.
- The paths to include files are listed, both for system include files and user include files (typically `.h` files). The `SYSTEMINCLUDE` and `USERINCLUDE` tags specify these, respectively. Note that individual `.h` files are not listed. These are derived from analyzing the source C++ files.

- The resource file used for this application is given by the `RESOURCE` tag.
- The system libraries that are needed to compile the code in this project are given by the `LIBRARY` tag.

Obtaining Unique UID Numbers

UID numbers for applications must be unique—that is, unique *across all applications that are written for Symbian OS*. During development, you can use certain development UIDs reserved for software development. These UIDs are in the range `0x01000000` through `0x0fffffff`. However, when you prepare software for distribution, it must have a unique non-development UID. To obtain one, send an email to the address `uid@symbian.com`, and title the message "UID request". State the number of UIDs you require and give your name with your return email address. Symbian will then allocate your UIDs and send them to you via email.

7.2.3 Building Configuration Files

Once we have an MMP project file, the next step is to construct a `bld.inf` file. This is used by system programs to indicate how many different project files there are. For the `helloworld` example, this file looks like this:

```
// BLD.INF
//
// Copyright (c) 2000 Symbian Ltd. All rights reserved.
//

PRJ_MMPFILES

HelloWorld.mmp
```

Most `bld.inf` files can have this simple format. In it, we are specifying that there is a single MMP file, and its name is `HelloWorld.mmp`. We can now generate the project files and make files using the `bldmake` command:

```
bldmake bldfiles
```

This will generate some further configuration files and a batch file called `abld.bat`. This batch file can be used to build a project

or create a makefile. The following command line will build an application for the emulator:

```
abld build wins
```

We can also use this batch file to generate construction files. Typing the command

```
abld makefile vc6 wins
```

will generate a project file for Visual C++ and a makefile for GNU C++. These files will be stored as

```
\Symbian\6.1\quartz\epoc32\build\MMP_path\project_name\wins.
```

The `makmake` command can be used to generate the project files in the local directory. It uses Perl to analyze the files specified in the MMP file and generate the individual project files. For our purposes, to use Visual C++, you should invoke the command as follows:

```
makmake helloworld vc6
```

This generates a project file for Visual C++ version 6, dumped into a file called `helloworld.dsw`. (N.B. You must be in the source code directory when executing this command.)

By executing the command

```
makmake helloworld wins
```

you will get a makefile generated in the file `helloworld.wins`. This can be used from the command line to make the example application.

7.2.4 Building the Application using Visual C++

Now let's build the application. We will first do this in Visual C++. Once Visual C++ is started, you should use the **File > Open Workspace...** menu selection and navigate to the `helloworld.dsw` file. Select this file, and the application and configuration files will be read by Visual C++. You should explore the class and file view panes. Notice that we can edit the configuration files we mentioned above right from the IDE.

The build process is started by selecting **Build > Build HELLOWORLD.APP**, or by pressing *F7*. The output from the build

sequence will be shown in the build output window, and is not terribly useful, except to scan for errors when they occur. All errors (compile errors and loading/linking errors) will show up in that output. Take a moment to glance over the output and be grateful you do not have to type those commands for each compilation.

Your application can then be executed by choosing either **Build > Execute** or **Build > Start Debug > Go**. Choose the latter if you want to debug the application. Visual C++ must first launch the emulator and will therefore ask you for the "Executable for Debug Session" through a dialog box. You should specify the Quartz emulator executable (a typical location is `\Symbian\6.1\epoc32\release\wins\udeb\Epoc.exe`), then click OK. The emulator will launch, and you will find the HelloWorld icon on the emulator's display. Click on this icon to launch the HelloWorld application.

7.2.5 Building the Application using the Command Line

Using Visual C++ to build and run applications is convenient when you want to use an IDE. However, there are many times when the command line is quicker and easier to use. As you have already created the command line makefile, building from the command line can be done in one command:

```
nmake -f helloworld.wins
```

Running the above command will produce a series of long command invocations, mostly of the Visual C++ compiler (`cl.exe`). Perl is used to convert the resource file, and the Visual C++ linker is used to put together the final application for the emulator to run. The end result will be the same as using the Visual C++ environment.

To launch the application from the command line, you need to type the name of the emulator executable (`\Symbian\6.1\epoc32\release\wins\udeb\Epoc.exe`). This will launch the emulator and you can select the application within that environment.

7.2.6 Building the Application for a Target Machine

To build the application for a real target machine, you must choose an architecture to compile for, out of the 22 possible. A typical

processor is the ARM processor; this can be given to `makmake` as below:

```
makmake helloworld arm4
```

This creates a makefile called `helloworld.arm4`, which can be used to make the application as follows:

```
nmake -f helloworld.arm4
```

Executing this command at the command line gives a tremendous amount of output, and, if all goes well, generates a compiled application in `\Symbian\6.1\Quartz\Epoc32\Release\urel\helloworld.app`.

At this point, delivery to a target machine requires the application, any libraries that the application uses, and any configuration files that assist the application. These all need to be placed in various places on the target machine's file system, and the SIS packaging system will help do this. Designing a `.pkg` file that directs where certain files go on the target machine, then packaging everything up with the `makesis` command, will produce a SIS file ready for installation.

7.3 Using the Emulators

7.3.1 Getting Started

The emulator for the Quartz reference design is distributed with the SDK for version 6.1 from Symbian. This emulator is fundamental in developing software for Symbian OS, so getting familiar with it is extremely important. The emulator comes in two variants: a release version and a debug version. The release version is faster and leaner, and suitable for running applications without debugging them. The debug version contains debug information useful to tools such as Visual C++, where it can be used for stepping though code, and testing various functions of the environment to check things like resource usage, memory leaks, and graphics manipulation.

You can start the emulator in two ways:

- Using the Windows "Start" button, you can burrow your way through menus to start it: **Start > Programs > Symbian 6.1 SDKs > Quartz > Emulator (debug)**. This will start the debug version of the emulator.

- You can execute the emulator program directly, either through the Windows Explorer or through command line invocation. The location on a default Symbian OS SDK installation is `\Symbian\6.1\Quartz\Epoc32\Release\wins\udeb\Epoc.exe`.

Figure 7.2 shows an example of the emulator in operation. It works like a generic Symbian OS Quartz device. Symbian OS licensees are allowed to customize the design and to change certain features of the operation to create versions of the SDK which are closer to their product's look and feel.

For design and testing purposes, the function keys have special significance to the emulator. The table below itemizes what they are used for:

Table 7.1 Function keys in the emulator

`F1`	Works with the application's menu system: starting menus or expanding menu items.
Alt `F2`	Simulates a "Help" key or function. Can be used for context-sensitive help on applications.
`F4`	Simulates switching removable media. Removable media are emulated by files in the host PC's directory structure. On the emulator, they can be referenced by the `Y:` drive designation.
`F5`	Simulates the opening and closing of the removable media door.
`F8`	Simulates a "record" button, starting and stopping a record operation.
`F9`	Simulates a "power on" condition, as if the "On" key were pressed.
`F10`	Simulates a "power off" condition, as if the "Off" key were pressed.
`F11`	Simulates the device's case being closed and opened.

Drives and their designations are important on the emulator. The default installation comes with a `C:`drive and a `Z:`drive. On a real Symbian OS device, the `C:` drive designates the memory-based default file system; this drive is emulated as part of the host PC's file system (`\Symbian\6.1\Quartz\Epoc32\wins\c` on a default installation). Likewise, on a Symbian OS device, the `Z:` drive designates the ROM file system, but is emulated as `\Symbian\6.1\Quartz\Epoc32\release\wins\udeb\z` on a default installation. Other device drive emulations may be set up; they are represented on the host PC as files. They can represent regular storage or components like removable media.

Other Aspects of the Emulator

There are too many interesting aspects of the Quartz emulator to include here. Check out the emulator guide in the Quartz documentation (see `\Symbian\6.1\Shared\CPP\Developer-Library\quartz\QuartzEmulator\emulator\index.html`). And, by all means, start up the emulator, and play! There is no substitute for getting your hands dirty.

7.3.2 Using the Emulator with Communication Devices

The emulator is an extremely useful tool for developing software. However, it is itself software by design—*emulating* hardware while not actually composed of hardware. When we develop communication software, we need to access communication devices. Since these are likely to be external devices not under Symbian's control, they will more than likely be actual hardware devices that need to be connected to the emulator.

So the emulator builds a bridge between software emulation and hardware peripherals. The default installation contains the following hardware connections:

- *Serial ports* for the emulator are connected to the host PC's serial ports. COMM::0 on the emulator is connected to COM1 on the PC; COMM::1 on the emulator is connected to COM2 on the PC.
- The *IR port* on the emulator must be an added device. Infrared port peripheral devices—such as a JetEye 7401 or an Extended Systems 9680B—can be connected to a serial port and can act as an IR port for a PC. Upon startup, the emulator will look for such a device, and will use it as the IR port for the Quartz emulator. The default port the emulator looks at is COM2.
- For *Bluetooth* functionality, the emulator looks for an Ericsson Bluetooth MiniKit (EBMK) connected to a serial port. By default, it looks on COM1 for the EBMK. If it finds a Bluetooth card there, the emulator assumes it has Bluetooth functionality and uses the EBMK to send and receive Bluetooth packets.
- *GSM access* is enabled by the emulator if it detects a GSM phone connected through an infrared port device on a serial port. The phone must be IR enabled (naturally) with the IR port activated. Once this arrangement is detected, the emulator will send and receive SMS messages directly through the phone.

Some of these configurations are customizable through configuration files. Specifically, the port used to detect the devices can be changed. Look in `\Symbian\6.1\Quartz\Epoc32\wins\c\system\data` for ESK files: `irda_wins.esk` to configure the IR-to-COM port map and `bt.esk` to configure the port for the EBMK.

Note that you cannot use a serial port to emulate more than one component. For example, you cannot expect to use the emulator's COMM::0 and a Bluetooth EBMK on the host PC's COM1. You must plan for device usage and specify which ports will be used through the `esk` files.

7.4 Introducing TodoX: A More Extensive Example

In the next several chapters, we will be using an example called `TodoX`. This is a simple example of how to exchange to-do list items between two devices. We will begin in the next chapter by using serial communications to exchange to-do list items and will enhance the example to handle new communications technologies as we introduce them from chapter to chapter. This section will include an illustration of how to install and run this software.

7.4.1 Building and Running the Application

As the first step, you must retrieve a copy of the software source from the Web site for this book. Access the URL ***http://www.symbian.com/books/socp/socp-support.html***. Navigate to the section on supporting software and download the file `todox.zip`. Follow the instructions on the Web site to unzip this archive and get the source files on your computer's disk storage.

Next, find the directory in which you stored the source files. Locate and examine the `bld.inf` file. This file is just as simple as in the `HelloWorld` example; it references the `todox.mmp` file as a project file.

Locate and examine the MMP file, shown below:

```
TARGET          TodoX.app
TARGETTYPE      app
UID             0x100039CE 0x101F3DCF
TARGETPATH      \system\apps\TodoX

SOURCE          TodoX.cpp TodoXdialogs.cpp
SOURCE          TodoXferBase.cpp
SOURCE          TodoXferSerial.cpp TodoXferSerialAO.cpp
SOURCE          TodoXferIRSerial.cpp TodoXferIRSocket.cpp
SOURCE          TodoXferBt.cpp
```

```
SOURCE          TodoXferEmail.cpp
SOURCE          AgendaFileReader.cpp

RESOURCE        TodoX.rss

USERINCLUDE     .
SYSTEMINCLUDE   \epoc32\include
LIBRARY         euser.lib efsrv.lib estor.lib etext.lib
LIBRARY         cone.lib apparc.lib
LIBRARY         eikcore.lib qikctl.lib eikcoctl.lib eikdlg.lib
LIBRARY         bafl.lib c32.lib
LIBRARY         agnmodel.lib agndasvr.lib agnapi.lib agnsynca.lib
LIBRARY         esock.lib irda.lib
LIBRARY         bluetooth.lib btmanclient.lib
LIBRARY         send.lib
```

From this file, we can deduce several things about `TodoX`:

- It is an application (see the `TARGETTYPE` line) with a UID of `0x101F3DCF` (I obtained this from Symbian for this project).
- The application and its supporting files are stored in the standard location of `\system\apps\TodoX` (the `TARGETPATH` line).
- There a large number of source files and quite a few more libraries that are needed (when compared to the `HelloWorld` example).
- There is a resource file (`TodoX.rss`) that supports the application.

The easiest way to build this application is to use the `bldmake` and `abld` command combination. Start by generating the construction files using the `bldmake` command in the directory to which you unzipped the source files, as below:

```
bldmake bldfiles
```

This should return to the command prompt with no output. However, the construction files should now be built, and you can build the application by giving the command below:

```
abld build wins
```

You should see much output as the application is compiled and installed.

Now, by running the Quartz emulator, you should see the `TodoX` icon (a generic one, with a small diamond shape on it). Click on this icon with the mouse and you should start the application. An example of how this might look is in Figure 7.3.

Figure 7.3 Running the `TodoX` example application.

7.4.2 Exchanging To-do List Items

The application displays a list of to-do list items that you have entered using the "To do" application. To start an exchange of items between computers, you need two PCs running the application. Follow these steps:

- Make sure the computers are connected. For a serial connection, connect the COM1 ports with a null modem cable.
- From the `TodoX` application on both PCs, click on the Mode menu from the menu bar and select the `Serial` choice (it is likely to be already selected).
- On the receiving computer, click on the `Receive` menu choice on the application's menu bar.
- On the sending computer, click on one of the to-do items in the list.

At this point, you should see some messages print on the blue status area on the bottom of the display screen. And you should see a new to-do list item listed on the receiving computer's application. Congratulations!

7.4.3 Editing the Source Files

You should use Visual C++ to work with the source files. To generate a Visual C++ project file, execute the command below:

```
abld makefile vc6
```

After command has run, you should have a file called `TodoX.dsw` in the source directory. Use this file as the workspace file in Visual C++.

7.5 Summary

In this chapter, we have reviewed the tools and procedures used to develop software for Symbian OS, and compiled and executed our first application. We reviewed the emulator environment itself, which provides a solid platform for application testing, and the way that the emulator interfaces with various hardware components connected to the host PC.

The next chapter begins looking at real communications programming by analyzing software development for serial communication. It is perhaps the most familiar for programmers and will serve as a good way to start discussing communications programming.

8

Serial Communications

It is always good when learning new material to start with what you know. It is a safe bet that most computer users have had some exposure to serial communication. For example, if you have ever transferred data between a handheld computer and a PC, then you have probably used serial communication. If you have ever used a modem on a PC, you have used serial communication. Serial communication is perhaps the oldest and the most familiar of data transfer functions. It is a good place to start as we introduce communications programming requirements and interfaces.

We will examine this area in the following way. We will first examine the *patterns* that are involved in communications programming. Across all programming platforms, common patterns emerge and we will analyze them. We will then use these patterns to look at the serial API for Symbian OS. We will also examine active objects as a crucial part of communications programming. Along the way, we will work through some examples and we will begin to look at vCalendar objects.

8.1 Programming Patterns for Communications

There is a set of patterns that emerge from the way that we work with communications programming. There are two sets of patterns that are important for our purposes here: those that develop from a local Symbian device perspective and those that apply to data exchange.

8.1.1 The Local Perspective

On the local Symbian device, there is a pattern to the way that communication is set up and executed. Specifically, communication works through five phases:

- *Initialization:* We start communication on the local Symbian device by initializing the communication system. On a larger, desktop computer, this is not an issue—the drivers and communication components are typically loaded at boot time by the system. However, on a small handheld device, only those servers and drivers that are immediately needed are started. Therefore, the operating system must be prepared before communication starts. This means loading drivers and starting servers as needed.

- *Opening the communication device:* In Symbian OS, the communication device interface is designed much like a file interface. For example, bytes are transferred by read and write operations. Just as the operating system sets up tables and structures when a file is opened, it also sets up its own internal housekeeping when we open a device. In addition, the system checks for access issues and sets up any protection mechanisms that must be in place for the application to use the device.

- *Configuring the communication device:* Once a communication device is open and access is granted, we must configure it. This could be as simple as telling the device to use settings in the communications database or it could involve setting parameters through function calls. There are very few settings for Bluetooth devices, for instance, while there are several for IR ports (e.g., baud rate and parity).

- *Exchanging data:* Now we finally get to the point where real data can be exchanged. To exchange information with another device, we engage that device using a protocol of some sort. All protocols have initialization sequences that must be worked through, with a (possibly lengthy) data exchange sequence, and a termination sequence.

- *Closing the communication device:* As a last step, the device that was used for communication can be closed. This is a signal for the OS to reset the device interface and to free up the resources that were dedicated to maintaining the device's operation.

As a demonstration of these patterns, let's dig into the `TodoX` example that we introduced in Chapter 7. I am going to build this example gradually. Figure 8.1 shows a complete set of the communication classes and their derivation relationships. There are seven example classes and two base classes (`CActive` is a Symbian OS class we will describe later). In the next few chapters, we will demonstrate the exchange of to-do list items over the serial port (in

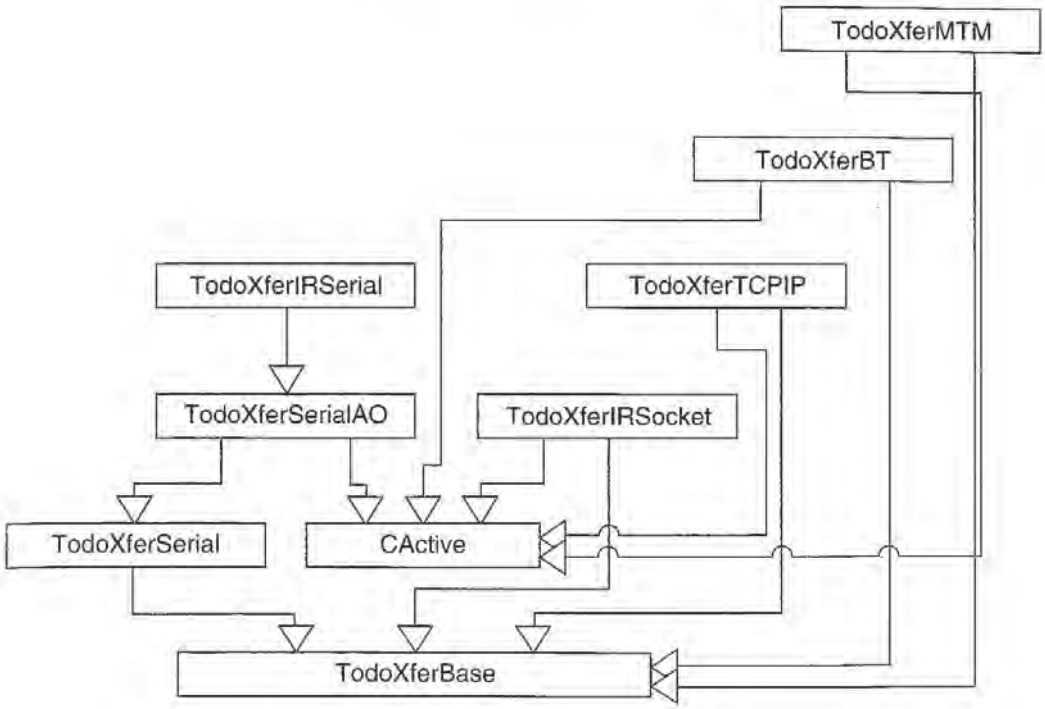

Figure 8.1 Diagram of the `TodoX` example communication classes.

two different ways), using serial IR, using TinyTP, with Bluetooth, and via email.

Coding for Examples

Beginning here and continuing throughout the rest of the book, you will find many code examples. These examples usually take the form of fragments of code—either real fragments grabbed from the middle of a definition or a complete function definition from a class. In all cases, the complete context of the fragment will be missing and only the minimal amount of code that is necessary is included.

For example, you will notice that class definitions will not include code from private sections or even protected sections. Since we cannot access these definitions, I will not include them.

The problem with this is that I have to make certain assumptions and take liberties with the code. For instance, you will notice that literal strings are expressed with the macro "`_L()`" when I should be using "`_LIT()`". Using the latter form, however, means including more context than I believe is relevant at the time. You will not find "`#include`" statements or macro definitions. I also will eliminate complete sections of irrelevant code by using comments to describe what that code actually did.

If at any time things get confusing, remember that all code examples are available on the support Web site for this book. The complete context is given in the source code and I have obeyed proper coding standards in that code.

The base class header that defines most of this structure is given below:

```
class TodoXferBase
{
public:
  TodoXferBase();
  ~TodoXferBase();

  virtual TInt Init(TDesC&)=0;
  virtual TBool OpenL()=0;
  virtual void Configure()=0;
  virtual void Close()=0;

  virtual void SendItem(CAgnEntry *)=0;
  virtual void ReceiveItem()=0;
  virtual void SendItemVCal(CAgnEntry *)=0;
  virtual CAgnEntry* ReceiveItemVCal()=0;

protected:
   void Send(const TDes8 &aText);
   void Receive(TDes8 &aText);
};
```

Note first that the first four functions implement four of the five phases we discussed for device communication. The next two—`SendItem()` and `ReceiveItem()`—implement the data exchange protocol. We have two functions that send and receive vCalendar objects. Finally, the `Send()` and `Receive()` functions will be at the heart of the underlying implementation of how to send strings between devices.

The first thing that we have to do is to decide on a protocol for this exchange—but for this introduction, let's assume we have already decided that detail. Further, let's assume that our application has been started. We have selected a serial transfer mode (appropriate for this chapter) and when we tap on a line representing a to-do list item, transfer will begin.

- Initialization will include loading the physical and logical device drivers and starting the communication server, if necessary. In addition, we need to establish a connection with the

communication server, and inform the server which communication module to load (recall CSY modules from Chapter 3).

- Opening the device is simple: one system call takes care of this step. We must make sure we field the return codes from the system call properly.
- For a serial port, there are several configuration parameters: baud rate, number of start and stop bits on the data frame, parity, and the handshake protocol. Each of these must be set before the to-do list item can be sent.
- Engaging in the protocol involves some initialization (to discover the remote device and to ensure it is ready), then the exchange of the item, then some signal for termination (basically a flag to announce that there are no more items coming).
- Finally, closing the device also means closing the connection to the communications server.

The remote device—receiver of the to-do list items—must go through a similar sequence of steps. The protocol is a bit different when receiving an item, but the rest of the steps are much the same.

Consider the following code:

```
void CTodoXAppUi::SendItem(TInt aIndex)
{
   // Select the chosen entry from the Agenda items
   iReader->ReadEntryL(iEntryIDs[aIndex]);

   // Start comm. phases based on mode of transfer
   switch (iXferMode) {
      case ESerialMode:
         iXferPort->Init(_L("ECUART"));
         iXferPort->OpenL();
         iXferPort->Configure();
         break;

       // ... other cases ...
}

// Now we engage the protocol: send the to-do item!
iXferPort->SendItem(iReader->iEntry);

     // close the device if necessary
     if (iXferMode == ESerialMode) iXferPort->Close();
}
```

We call this function from the AppUi class of the application once a list item has been tapped with the stylus. Notice that the sequence works out in the order we have described it above. The

variable `iXferPort` points to different port implementations, but the pattern from initialization through device closing is the same. For example, if `iXferMode` has the value `EBtMode` (indicating Bluetooth data transfer), then iXferPort will still point to a device object (in this case, a `TodoXferBt` object, but subclassed from the same base class). We would use the same sequence of function calls as in the sequence above.

Receiving an item is basically the same:

```
void CTodoXAppUi::ReceiveItem()
{
   CAgnEntry *iEntry;

   // Start comm. phases based on mode of transfer
   switch (iXferMode) {
      case ESerialMode:
         iXferPort->Init(_L("ECUART"));
         iXferPort->OpenL();
         iXferPort->Configure();
         break;

      //... other cases ...
   }

   // Now we engage the protocol: receive the to-do item!
   iXferPort->ReceiveItem();

   // Close the port if necessary
   if (iXferMode == ESerialMode) iXferPort->Close();
}
```

8.1.2 Between Symbian Devices

Between two Symbian devices, there is also a pattern that develops in the protocol that controls the data exchange. There is always a protocol that is used, even when the function used to exchange data is made up by the programmer. Protocols are, by definition, the arrangements that both sides go through to get the data across. Whether it is a standard protocol or one that a programmer invents, a protocol is always used nonetheless.

Protocols typically have three phases:

- *Establishment:* The two sides must establish and initialize a connection before any data is exchanged. Each side will initialize data, allocate resources, and otherwise prepare for the exchange. In addition, each side works with the other to get to the right state with respect to the data exchange.

- *Data exchange:* This is the core of a protocol, the phase where real data flows between the two sides. Issues such as flow control and error checking are important here, as well as the format of the data and the methods used for exchange.
- *Termination:* In this phase, the protocol shuts down. In the best case, it does so gracefully, informing each side of the impending shutdown and allowing for error checking and the releasing of resources. In the worst case, timeout occurs and the protocol is aborted in whatever state it was in.

The protocol sequence in Figure 8.2 depicts the protocol for our to-do list item exchange example. It is admittedly simplistic, but it serves our purpose. The top tier is the establishment phase, where each side announces it is ready and the sender informs the receiver how many items are coming. The middle tier is the data exchange phase, where each item is sent in three parts: priority, due date, and item text. The bottom tier is the termination phase, where each side announces it is done, and they terminate the conversation. We will discuss this protocol later in more detail.

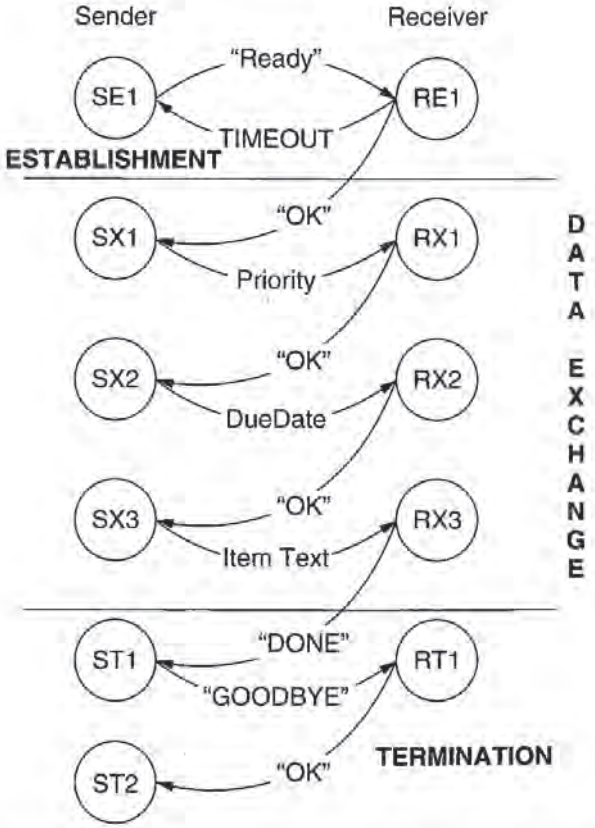

Figure 8.2 The protocol sequence for the to-do list item example.

It might not be easy to see the worth of these patterns now. They appear quite simple. As we progress through the Symbian OS communications architecture, we will see how these patterns appear again and again. Understanding them will greatly help in understanding how the communications architecture works.

8.2 The Communications Server

Using communications in Symbian OS begins with the communications server. The communications server functionality is implemented by the `RCommServ` class. The typical procedure to use a communication device is as follows:

1. Create an instance of the `RCommServ` class.
2. Connect to the communications server. Instances of the `RCommServ` class can access the connector function:

   ```
   TInt Connect();
   ```

   This function will connect the application to a running communications server.
3. When a successful `Connect()` call has been made, the `RCommServ` object can be used in all subsequent API calls that require it. Most initial `Open()` calls to start using a communications device require a `RCommServ` object.
4. When communication activity is completed, the connection with the communications server must be severed with a call to `Close()`:

   ```
   TInt Close();
   ```

There are other functions defined for the `RCommServ` class, but we will encounter these as we need them.

Sometimes, to connect to the server, we must make sure that it is started and running. The way to start the server is to use a utility function (found in `c32comms.h`):

```
TInt StartC32();
```

This call starts the communications server. If the server is not running, it is started. If it is already running, nothing happens. It is not an error to try to start a running server.

8.3 The Serial API

So, let's dig in and take a look at the serial communication API. As a guide, we will use the patterns we outlined in the previous section.

As an aside, note that there will be two versions of the serial code developed for the `TodoX` example. The one shown in this section demonstrates how to use the serial port without active objects. Later in this chapter, we will reexamine the example and rewrite the code to use active objects.

8.3.1 Initializing the System

Step 1: Load the drivers

There are several steps to initializing the serial interface. As a first step in the process, the drivers must be loaded. This means both the physical and the logical drivers. Two functions from the `User` class library apply here:

```
static TInt LoadPhysicalDevice(const TDesC& aFileName);
```

This call causes the OS to load the device driver for the physical device given as the argument to the function. Notice the parameter is a string, the name of the file containing the driver. The call returns a code to reflect the success or failure of the device loading.

```
static TInt LoadLogicalDevice(const TDesC &aFileName);
```

This call causes the OS to load the device driver for the logical device given as the argument to the function. Again, the parameter is a filename, and the call returns a code to reflect success or failure.

The name given for the driver file is assigned by Symbian and must be present in defined places in the file system. The code fragment below explains the name assignments

```
#if defined (__WINS__)
_LIT(PDD_NAME, "ECDRV");
_LIT(LDD_NAME, "ECOMM");
#else
_LIT(PDD_NAME, "EUART1");
_LIT(LDD_NAME, "ECOMM");
#endif
```

The definitions are different, depending on your environment. The symbol `__WINS__` is automatically defined if you are compiling code for the emulator environment.

Note here that the filenames have no suffix; the OS automatically adds the suffix when needed (`.PDD` or `.LDD`). Also, the filenames are not specified with an absolute pathname; the system will search for the proper driver files (usually in `\System\Libs` on all drives present).

Step 2: Connect to the communication server

The next step is to connect to the communications server. We should start the server and connect to it as described in Section 8.2.

Step 3: Load the serial CSY module

To interface the communications server with the hardware, we must load the appropriate CSY module. In the serial case, this is the `ECUART` module, and loading is straightforward. Again, we use an `RCommServ` instance to call

```
TInt LoadCommModule(const TDesC &aFileName);
```

This call again uses a string filename and causes the communications server to load the CSY module specified. As with loading drivers, the system will automatically append the suffix (`.CSY`) and will commence a system-wide search for the drivers specified, so no absolute pathname is needed.

Step 4: Initializing in the to-do list example

Before we can initialize the serial port, we must have an `RCommServ` object. In the `TodoX` example, the `TodoXferSerial` class declares this object, as well as other objects and functions we will need.

```
class TodoXferSerial: public TodoXferBase
{
public:
   TodoXferSerial(CTodoXAppUi*);
   ~TodoXferSerial();

   TBool OpenL();
   TBool OpenL(TDesC&);
   TInt Init(TDesC&);
   void Configure();
   void Configure(TBps baudrate, TDataBits databits,
      TParity parity, TStopBits stopbits);
   void Close();
```

```
    void SendItem(CAgnEntry *);
    void ReceiveItem();
    void SendItemVCal(CAgnEntry *);
    CAgnEntry* ReceiveItemVCal();

private:
    void Send(const TDes8 &aText);
    void Receive(TDes8 &aText);

protected:
    RCommServ server;
    RComm commPort;
    CTodoXAppUi* iAppUi;

};
```

Initializing the serial port in the `TodoX` example is done through the `Init` function of the `TodoXferSerial` class:

```
TInt TodoXferSerial::Init(TDesC &aModuleName)
{
     TInt result;

    // Load the physical device driver
    result = User::LoadPhysicalDevice(PDD_NAME);
    if (result != KErrNone && result != KErrAlreadyExists)
       User::Leave(result);

    // Similarly for the Logical device driver
    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 comm server
    result = server.Connect();
    if (result != KErrNone) User::Leave(result);

    // Load the comm module
    result = server.LoadCommModule(aModuleName);
    User::LeaveIfError(result);

    return ETrue;
}
```

Note that the calls to load device drivers will return two acceptable result codes. These are either `KErrNone`, for no error, or `KErrAlreadyExists`, for the case that the drivers are already

loaded. All other result codes cause the code to abort through `User::Leave()`. This is also the case with starting the communications server: acceptable result codes indicate no error or an existing running server.

8.3.2 Opening the Serial Port

Opening the serial port is straightforward, but requires an instance of the `RComm` class. An `RComm` class object represents a hardware port and lies at the base of the protocol stacks. Opening a port uses one of the following `RComm::Open` functions:

```
TInt Open(RCommServ &aServer, const TDesC &aName,
   TCommAccess aMode);
TInt Open(RCommServ &aServer, const TDesC &aName,
   TCommAccess aMode,TCommRole aRole);
```

Both functions require a communications server object, the name of the port to open, and the mode used to access that port. The name of the port is a string, chosen from the names assigned to ports by the port drivers. In version 6.1 of Symbian OS, there are two serial ports, called `COMM::0` and `COMM::1`. The access mode is chosen from the enumeration type `TCommAccess`:

```
enum TCommAccess {ECommExclusive,ECommShared,ECommPreemptable};
```

The port may be opened for exclusive use by the current application, may be shared with other applications, or may be "pre-emptable", that is, given up upon request by another application.

The second version of the call allows the caller to configure the role of the port either as a DCE (the communication initiator) or a DTE (the communication receiver):

```
enum TCommRole {ECommRoleDTE, ECommRoleDCE};
```

The result code from the `Open` call will indicate the status of the open operation. A result code of `KErrLocked` indicates that the port is in use and opened by another application.

Back to our `TodoX` example, we can use the `RComm` object declared for the `TodoXferSerial` class to open the serial port:

```
TBool TodoXferSerial::OpenL()
{
   TBuf16<25> portname;

   portname.Copy(_L("COMM::0"));
   return OpenL(portname);
}
```

```
TBool TodoXferSerial::OpenL(TDesC &aPortName)
{
   TInt result;

   result = commPort.Open(server,aPortName,ECommExclusive);
   User::LeaveIfError(result);

   return ETrue;
}
```

Note that we want exclusive use of the port for the duration of the to-do item exchange.

There is another way to open the port that an `RComm` object represents. Specifically, this uses the `OpenWhenAvailable` call and is designed more for active objects. We will elaborate further when we discuss active objects later in the chapter.

8.3.3 Configuring the Serial Port

Once the port is open, we must configure it. The more abstractly we view the hardware, the easier that hardware is to configure. Unfortunately, we do not have a very abstract view of the serial port; we are, in fact, dealing directly with the port through the communications server as we defined before. So we have several details to work with.

For serial ports, we use two calls from the `RComm` class to get and set their configuration:

```
void Config(TDes8 &aConfig) const;
TInt SetConfig(const TDesC8 &aConfig);
```

The argument to both calls is a `TDesC8` string declaration, which in this case is more accurately viewed as an array of 8-bit quantities. Each of those bytes contains a configuration parameter. Through the magic of C++ typing, we can view this byte array as a structure through the `TCommConfigV01` class:

```
class TCommConfigV01
{
public:
  TBps iRate;
  TDataBits iDataBits;
  TStopBits iStopBits;
  TParity iParity;
  TUint iHandshake;
  TUint iParityError;
  TUint iFifo;
  TInt iSpecialRate;
```

```
    TInt iTerminatorCount;
    TText8 iTerminator[KConfigMaxTerminators];
    TText8 iXonChar;
    TText8 iXoffChar;
    TText8 iParityErrorChar;
    TSir iSIREnable;
    TUint iSIRSettings;
  };
```

Each of these variables represents a configuration parameter for an RComm object. Rather than fill these pages with the definitions of each one, you should review them in d32comm.h. When an RComm object is constructed, each of these is given a default value.

A good example of how to configure the port is given in the TodoX example. Consider the Configure function from the TodoXferSerial class:

```
void TodoXferSerial::Configure(TBps baudrate,
                               TDataBits databits,
                               TParity parity,
                               TStopBits stopbits)
{
  TCommConfig cBuf;
  TCommConfigV01 &c=cBuf();

  // Get the current configuration
  commPort.Config(cBuf);

  // Set new settings
  c.iFifo = EFifoEnable;
  c.iRate = baudrate;
  c.iHandshake = KConfigObeyCTS;
  c.iTerminatorCount = 0;
  c.iDataBits = databits;
  c.iParity = parity;
  c.iStopBits = stopbits;

  // Write the settings out
  commPort.SetConfig(cBuf);
}
```

First, notice that we use the TCommConfig structure to get and set the configuration. The TCommConfig structure is the byte array; the TCommConfigV01 structure is the class we use to manipulate each element in the byte array by name. Again, by taking advantage of C++ type casting rules, we can align the memory areas of these two structures and make them work together as one.

In the above example, we first get the configuration, then adjust it and set it again. This saves code, because many of the settings

are correct already and we only want to change a few of them. We could start with an uninitialized `TCommConfig` structure, especially if there were many parameters to change. But when we only need to alter a few settings, copying the old configuration works well.

8.3.4 Exchanging Data

We are now (finally!) ready to exchange data with the other computer. At this point, we will assume that the other side has gone through the same steps that we have and is waiting for data from the sender.

Data is received and sent through the `Read()` and `Write()` functions and their variations from the `RComm` class. We will look at these operations separately. As we look at the functions that implement reading and writing, it is important to remember that reading and writing data are *asynchronous* operations. This means that these operations might have to wait for data to arrive, and this waiting will cause blocking in the execution of the application.

Reading data

Reading data from the serial port is done through various forms of the `Read()` function from the `RComm` class. There are five forms of `Read()` we will consider:

```
void Read(TRequestStatus &aStatus, TDes8 &aDes);
void Read(TRequestStatus &aStatus, TDes8 &aDes,
   TInt aLength);
void Read(TRequestStatus &aStatus,
   TTimeIntervalMicroSeconds32 aTimeOut,TDes8 &aDes);
void Read(TRequestStatus &aStatus,
   TTimeIntervalMicroSeconds32 aTimeOut,TDes8 &aDes,
   TInt aLength);
void ReadOneOrMore(TRequestStatus &aStatus,TDes8 &aDes);
```

Each form reads data through the `RComm` object and places that data in the `aDes` buffer given as the argument. This buffer is of type `TDes8`, which means it is an array of bytes. All communication—whether through the emulator or from real hardware—is done in 8-bit quantities.

Each form also contains an argument of the type `TRequestStatus`. The operations attempt to read through their `RComm` object until they complete, at which point the `TRequestStatus` object changes its value to reflect the status of the operation. This is the way asynchronous operations create result codes; they cannot return codes immediately as synchronous functions do.

The first two variations reflect differences in the amount of the input data stream and the length of the input buffer. The first form will read data until the buffer given as the second argument is full. The second form will read `aLength` bytes and return whether the buffer is full or not. The third and fourth forms add a timeout feature to the first and second implementations. In addition to filling the buffer, they will timeout after `aTimeOut` microseconds, and return a buffer filled with whatever data was obtained in that time interval. Should the buffer fill up during those `aTimeOut` microseconds, the functions will cancel the timer and return.

The last form is a special one in that it examines the port hardware and returns whatever it can. If the serial driver's buffer contains data, this function will retrieve that data up to the length of the input buffer. If there is no data waiting, then the function blocks until the first byte is delivered. It then returns this byte.

Because reading is an asynchronous operation, we must be able to wait until it is done. We can use one of several functions from the `User` class to accomplish this:

```
static void WaitForRequest(TRequestStatus &aStatus);
static void WaitForRequest(TRequestStatus &aStatus1,
                           TRequestStatus &aStatus2);
static void WaitForAnyRequest();
```

Each function blocks execution, waiting for the status variable to change its value. When the value changes, the function returns and execution resumes. The first version waits for one status variable; the second version waits for two; the third will return on a change to any status variable.

Writing data

Writing data follows much the same pattern as reading. The write operation is also asynchronous; it blocks until it can complete its operation and works in conjunction with a `TRequestStatus` variable.

There are four forms of the write operation:

```
void Write(TRequestStatus &aStatus,const TDesC8 &aDes);
void Write(TRequestStatus &aStatus,const TDesC8 &aDes,
   TInt aLength);
void Write(TRequestStatus &aStatus,
   TTimeIntervalMicroSeconds32 aTimeOut,
   const TDesC8 &aDes);
void Write(TRequestStatus &aStatus,
   TTimeIntervalMicroSeconds32 aTimeOut,
   const TDesC8 &aDes, TInt aLength);
```

Each form sends data out through an RComm object; each sends an eight-bit buffer that was given as a parameter; each sets its status variable according to the status of the write operation. Two forms (the second and fourth) send a specific amount of data from the buffer. Two forms (the third and fourth) use a timer to wait aTimeOut microseconds for the write operation to succeed.

More on waiting for operations

The asynchronous nature of I/O has positive and negative facets. While it makes reading and writing more complicated, it also provides an opportunity for the processor to use the waiting time for other things (it may even power down the Symbian device during this time).

The behavior of the I/O operations is as follows: each function will return immediately, regardless of the completed status of the operation. Only when the TRequestStatus variable has an integer value of zero (KErrNone) is the data in the operation's buffer valid. Until that point, you are free to execute other code. The WaitForRequest function calls simply save you the execution cycles of spinning in a loop constantly checking the value of the status variable.

Because the I/O operation is executed in parallel, it can be cancelled. Given that the code is not blocked in a WaitForRequest function, one of the RComm functions below can be called:

```
void ReadCancel();
void WriteCancel();
```

These cancel Read and Write operations, respectively.

Reading and writing in the to-do item example

Reading and writing serial data in the TodoX example is accomplished through two low-level calls:

```
void Send(const TDes8 &aText);
TBool Receive(TDes8 &aText);
```

These functions operate directly on the serial interface and are called by SendItem() and ReceiveItem(), which implement the item exchange protocol.

The Send() function illustrates an issue we have looked at before: data frame format. Even sending a string—a sequence of

bytes—over a simple serial port interface requires a minimal data frame. The issue is string length: how do you indicate the length of the string that is to be sent? Our data frame shown in Figure 8.3 includes the length of the string with the bytes that compose the string's value as the first byte of the frame.

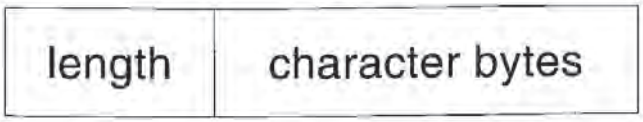

Figure 8.3 A simple data frame format.

This is a simple data frame. It informs the remote receiver of the number of bytes the sender is sending. Since the receiver does not get this information any other way, sending it this way is important.

The code of the `Send()` function is below:

```
void TodoXferSerial::Send(const TDes8 &aText)
{
  TInt len;
  TRequestStatus status;
   TBuf8<2> buffer;

   len = aText.Length();

   // Send the length
   buffer.SetLength(0);
   buffer.Append((TChar)len);
   commPort.Write(status, buffer, 1);
   User::WaitForRequest(status);
   User::LeaveIfError(status.Int());

   // Send the text
   commPort.Write(status, aText);
   User::WaitForRequest(status);
   User::LeaveIfError(status.Int());
}
```

Note that even when we are sending an integer, it is captured as a byte and sent—the easiest way to do this is as appended to an eight-bit buffer of characters. Further, note that even though we can send any size of string this way (up to the bounds of the `TInt` type), Agenda application to-do entries can only be up to 255 characters long.

The `Receive()` function is the counterpart of the above sending code. It receives an integer as a character, decodes the value, then pulls in that many bytes:

```
TBool TodoXferSerial::Receive(TDes8 &aText)
{
   TInt len;
   TRequestStatus status;
   TBuf8<2> buffer;

   // Read the length
   commPort.Read(status, buffer, 1);
   User::WaitForRequest(status);
   User::LeaveIfError(status.Int());

   // Read the text character by character
   len = (TInt)buffer[0];
   if (len <= 80) {
      commPort.Read(status, aText, len);
      User::WaitForRequest(status);
      User::LeaveIfError(status.Int());
   } else {
      len = 0;
   }
   return ETrue;
}
```

Extend `TodoX` with `Expect`

A useful function for the `TodoX` example would be one called `Expect`. This function would take a string (a `const TDesC8` argument) and would read from the `commPort` until that string was found, or until a specific time has elapsed.

Write this function for the `TodoX` example. You should be able to implement the timing without needing any kind of timer if you do not use `ReadOneOrMore()`. However, you might want to use `ReadOneOrMore()` for this function, and then implementing a timeout would not be possible. Check the `CTimer` class for this. This will make you appreciate using active objects (see a later section).

8.3.5 Closing the Port

When all is done, the termination of communication occurs in two steps. First, we close the port. Second, we close our connection to the communication server. Each operation is handled by a call to

```
void Close();
```

from the `RHandleBase` class, since both `RCommServer` and `RComm` are descendants of this class.

In the to-do list item example, closing the port is handled by the `Close()` function from the `TodoXferSerial` class:

```
void TodoXferSerial::Close()
{
   commPort.Close();
   server.Close();
}
```

8.4 Other Interface Calls

There are many other calls associated with the `RCommServ` and `RComm` classes. We will get to many of them eventually, but we will highlight a few more of them here.

8.4.1 RCommServ

There are several calls of interest here, highlighted below:

```
TInt UnloadCommModule(const TDesC &aName);
```

This call is the reverse of the `LoadCommModule()` call we used above in the initialization of the communication server. While `LoadCommModule()` will load a CSY module, `UnloadCommModule()` will remove the CSY module from the kernel's tables. Unloading may be desirable if kernel resources are scarce.

```
TInt NumPorts(TInt &aNum);
```

This call returns the number of ports served by the communication server. These will usually include at least the serial port and the IR port.

```
TInt GetPortInfo(const TDesC &aName, TSerialInfo &aInfo);
TInt GetPortInfo(TInt aIndex, TDes &aModuleName,
   TSerialInfo &aInfo);
```

These calls retrieve information from the communication server about the ports that it serves. Information may be retrieved by port name (the first call) or port number/module (the second call). In either case, the call returns the information in the form of the last parameter—a `TSerialInfo` object, defined as below:

```
class TSerialInfo
{
public:
  TPortDescription iDescription;
  TPortName iName
```

```
  TUint iLowUnit;
  TUint iHighUnit;
};
```

This class contains a description of the serial port (in terms of CSY modules), the name of the port, and the device number range.

8.4.2 RComm

The RComm class has many functions defined for it. There are a few more of interest for us at this stage.

```
void OpenWhenAvailable(TRequestStatus &aStatus,
   RCommServ &aServer, const TDesC &aName)
void OpenWhenAvailable(TRequestStatus &aStatus,
   RCommServ &aServer, const TDesC &aName, TCommRole aRole)
```

These calls are a variant of the Open() call, and work differently from the Open() call when the port requested is not available. Instead of returning an error code, these calls block and wait for the port to become unused. When the port becomes available, these calls will open the port, setting the status variable, and return.

```
void OpenWhenAvailableCancel();
```

This call will cancel the operation that is begun when the OpenWhenAvailable() functions block and wait for a port. Because of their heavily asynchronous nature, these calls are more suited for active objects than for straight usage. We will discuss active objects in the next section.

```
TInt QueryReceiveBuffer() const;
```

This call checks how much data is waiting in the receive buffer to be consumed by the next Read call.

There are also several functions that are used to notify an application when an I/O condition exists. The system can notify an application when a port's configuration changes, for example, or when data arrives. These calls are asynchronous and must be waited for. The status variable setting indicates the notification condition. These functions are accompanied by a companion function which cancels the notification condition. We will encounter these later on.

8.5 Active Objects

If "the ends justifies the means", as it is said, then the `TodoX` example works just fine. The end result is that a to-do list item is transferred between devices. However, the means by which this is carried out is not desirable. In the current implementation of the example, the application is essentially suspended while each data communication is executed. This could get quite annoying if you were to transfer many items between devices. You can see this if you run the `TodoX` example on the emulator and watch closely: the status messages appear *out of sync* with the operation they report on (they should appear before that operation takes place but they actually appear afterwards).

A solution to this would be to use a threaded application. One thread would handle the communication I/O and another would handle the user interface interaction. In fact, the best solution here would be one implemented by the OS—even in the kernel—because of the notification issues and thread suspension that must go on.

Active objects represent Symbian OS's solution to this issue. An active object is a thread that handles event notifications from the kernel. We are specifically interested in device-oriented events, e.g., the arrival of data in a buffer or the completion of an I/O operation. There are also other events that are meaningful to communications, e.g., an alarm event when a timer counts down to zero, indicating timeout.

Active objects are derived from the `CActive` class, which provides access to a `CActiveScheduler` object. An active object must implement at least two functions:

```
void RunL();
void DoCancel();
```

The `RunL()` function is the heart of an active object implementation. Upon construction, the active object will create and initialize anything it needs to. When a function is called that performs an asynchronous operation that might involve waiting, instead of calling `User::WaitForAnyRequest()`, the active object needs to call `SetActive()`, inherited from the `CActive` class. This suspends the active object thread, turning over control of the asynchronous operation to a scheduler. When this operation completes and generates an event the active object has registered for, the scheduler calls the `RunL()` function for the active object.

The `DoCancel()` function must be implemented to cancel the actions implemented by the active object. On receiving a request

to cancel the operations for an active object, the system will first check whether there is an outstanding request to this object, then will call `DoCancel()`.

8.5.1 State Diagrams

To effectively use an active object, especially with communications, we need to develop a *state diagram*. A state diagram is a depiction of the states that an application can be in and how it moves between those states. These states are usually defined—for our communications purposes, that is—by the times that our application is waiting for a communication event.

Consider Figure 8.2 again. The circles can represent states; the "S" states are ones a sender can be in and the "R" states are ones that a receiver can be in. We can express a state diagram by looking at just the sender or just the receiver side.

The states of communication are determined by the actions of waiting for a communication event. So, each Read or Write, for instance, should instigate a new state. This means we have to split up the states from Figure 8.2 into those that wait for `Read` operations to complete (the "S" states) and those that wait for `Write` operations to complete (the "R" states).

Figure 8.4 contains the new state diagram for a to-do item sender. It depicts all the states a sender can be in, and the ways we move between the states. A `SError` state has been added, and a special designation is given to the last state—`ST2`. That state is a "final state", that is, if we are in that state when input is complete, we have correctly traversed the state diagram. Otherwise, we have to flag an error.

8.5.2 Translating State Diagrams to Active Object Code

Once we have a state diagram, we can produce the code for the active object.

We must first have a way to start the active object. This is a function that initializes the active object and executes the first I/O operation. This function passes control to the thread scheduler by using the function call below:

```
void SetActive();
```

A call to this function relinquishes control of the thread to the scheduler and allows the system to wake up the thread when the I/O operation is complete.

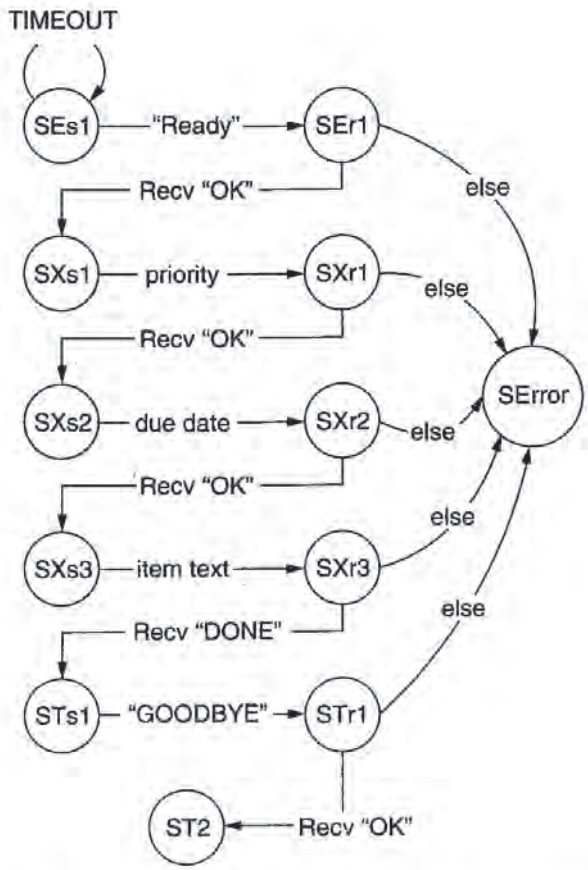

Figure 8.4 State diagram for the sending side of the TodoX example.

In the TodoX example, this is the implementation of SendItem for a new class, TodoSerialXferAO. This function is shown below:

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

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

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

   // Start the protocol
   iAppUi->SetProgress(_L("Starting the protocol"));
   buffer.Copy(KReady);
   Send(buffer);

   iSendingState = ESEs1;
}
```

Notice the differences between this implementation and the one for `TodoXferSerial`. First, many of the variables are now global to the class. This is because the operation extends over the whole class now, not just one function. Second, we use a new variable—`iSendingState`—to remember the state we are in. Finally, notice the lack of `User::WaitForAnyRequest()` calls. The system is doing the waiting for us. This is more obvious by the new implementation of Send():

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

   len = aText.Length();

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

   SetActive();
}
```

There are no `User::WaitForAnyRequest()` calls and there is a call to `SetActive()` at the end of the function.

Once we have started the active object and turned over control to the scheduler, we need a way to continue its execution whenever I/O operations complete. This is found in the implementation of the `RunL()` function. This function is essentially one big `switch` statement, based on the value of the state variable. Each case in the `switch` statement looks something like this (from `TodoXferSerialAO`):

```
case ESEr1:
  if (status == KErrNone) {
     if (buffer != KOk) {
        iSendingState = ESError; // indicate an error
     } else {
        iAppUi->SetProgress(_L("Sending the priority"));
        buffer.Format(KPriorityFormat, priority);
        Send(buffer);
        iSendingState = ESXs1;
     }
} else {
   iSendingState = ESError; // indicate an error
   iAppUi->SetProgress(KRcvErrMessage);
}
break;
```

At the beginning of each case, we can assume that the previous I/O operation has completed with a value status variable. We check the status variable to determine if the operation completed successfully. If it was successful, we process the results, engage another I/O operation, and change the state variable for the next invocation of `RunL()`. If it has not completed successfully, we must deal with it somehow—perhaps aborting the active object or resetting the communication stream. At the end of the case, if we want to continue, we must call `SetActive() again`.

8.5.3 Using an Active Object

Using an active object is not much different from using a sequential communication object. However, there are a few subtle differences:

- Active objects are typically initialized in addition to being instantiated. This means that we need a `new` operator to get an active object, then use initialization code to set the object up. In the `TodoX` example, this is accomplished by `TodoXferSerialAO::NewLC()`:

```
TodoXferSerialAO* TodoXferSerialAO::NewLC(CTodoXAppUi*
aAppUi)
{
   TodoXferSerialAO* self=new (ELeave) TodoXferSerialAO(aAppUi);

   CleanupStack::PushL(self);
   self->ConstructL();

   return self;
}
```

  This code calls the `ConstructL()` function in addition to instantiating a `TodoXferSerialAO` object. The main thing `ConstructL()` does is to add the current active object to the scheduler:

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

  This code could have done other operations. This is much like an object's constructor, but it must be run after object instantiation.

- When the active object returns, nothing can be assumed about the completion of the I/O operation. In the `TodoXferSerial` class, we could assume that, when `SendItem()` returned, the

item was sent to a receiver. However, in the `TodoXferSerialAO` class, we can only assume that the transfer was *started*, not completed. This is why we close the communication port only if we are using the `ESerialMode` mode to transfer items and allow all others to close the port themselves when they are done.

- There must be a way for the active object to signal to the calling class that it is done performing an operation. As we discussed before, the active object is asynchronous, and therefore cannot simply return a value to a calling object. It requires a callback function from the calling object. In the `TodoX` example, this is handled by the `ProcessReceivedItem()` function, which deletes the active object and enters the to-do item on the display.

Further Reading on Active Objects

For more information on active objects, there are two references that are extremely valuable. *Professional Symbian Programming*, by Martin Tasker, *et al.*, gives a good introduction to active object programming. Also, the Active Object sections in the Symbian OS version 6.1 system documentation is very good. You can find this by using the Programs > Symbian 6.1 SDKs > Quartz > C++ SDK Documentation menu sequence from the Windows Start button.

8.6 A Note About vCalendar Objects

Before leaving this chapter, note that the `TodoX` example is a bit contrived. I arbitrarily chose three properties of a to-do list item and worked them into a simple transfer protocol. There are, in fact, many other properties—like alarm settings, for instance—that need to be included in a complete to-do list item transfer implementation.

In fact, we can completely represent a to-do item by using a vCalendar. As we stated back in Chapter 5, vCalendar objects are designed as objects with a standard format, so that different platforms and operating systems can share them. Consider Figure 8.5, which contains a representation of a to-do item from the list shown in Figure 7.3.

The Symbian OS implementation of vCalendar objects includes the functionality needed to transfer them between devices. I will cover this in later chapters.

```
BEGIN:VCALENDAR
VERSION:1.0
BEGIN:VTODO
UID:2
DESCRIPTION:Eat lunch with Bob
DUE:20010706T000000Z
PRIORITY:2
X-EPOCTODOLIST:To-do list
X-EPOCAGENDAENTRYTYPE:TODO
CLASS:PUBLIC
DCREATED:20010706T000000Z
LAST-MODIFIED:20010831T091100Z
CATEGORIES:X-65536
END:VTODO
END:VCALENDAR
```

Figure 8.5 An example vCalendar representation of a to-do list item.

8.7 Summary

This chapter has initiated the section on communications programming by addressing serial communications. We covered several important topics:

- We started by examining the patterns that are found in communications programming. There are five stages to programming the local device, and three stages to engaging the remote device in a data exchange protocol. Along the way we introduced an example: `TodoX`, which exchanges to-do list items between devices via the serial port.
- We then took a look at the API for communication using the serial port. Instead of an exhaustive itemization, we looked at the API functionally, itemizing what was needed for programming.
- We defined active objects as a way to implement threaded communications using the kernel. Active objects make applications more responsive by parallelizing them, establishing a thread for the I/O operations that might cause an application to halt temporarily while waiting.

- Finally, we mentioned vCalendar objects as a way to upgrade from the `TodoX` example.

The next chapter will tackle IR communications. At first, IR looks like serial communications through a new port. However, we will introduce sockets that will enable us to reach into higher level protocols.

9

Communicating via Infrared: Serial Communication

Sir William Herschel is credited for discovering the infrared region of the electromagnetic spectrum. His son, Sir John Herschel, demonstrated in 1840 that infrared light existed by showing that different light had different rates of absorption and transmission. Early uses for IR were from astronomy (examining infrared light sources) to the military (lasers). It took until 1916 for infrared light to be used to transmit data, and then it was used to transmit secret messages during war. Infrared capabilities of one form or another have shown up on communication devices since the 1980s.

The infrared port on a Symbian device represents a transition of sorts between the serial communication of the last chapter and the more complex protocol structures we will tackle in future chapters. It can be programmed at the raw data level of a serial port and at the higher level of an upper layer protocol. We will let this transitory nature guide the way we discuss IR: this chapter will deal with serial IR programming, and we will wait to discuss upper level programming until Chapter 11.

This chapter will be relatively short. We will first pause to review how serial IR programming fits into the Symbian OS communications architecture. We will then dig into how to program the IR port as a serial device. We will complete this chapter with a look at sending vCalendar objects across a serial IR interface.

9.1 Admiring the Landscape

Serial IR programming gives us a great opportunity to pause for a moment and admire the way the Symbian OS communications infrastructure is designed. Converting an application from serial port communication to serial IR communication is simple and

requires little code alteration, because of the modularity built into the architecture.

Consider the module/driver structure that develops when using serial port communication. By the time we are able to correctly call `Read()` and `Write()` functions from the `RComm` class, we have a structure much like the one in Figure 9.1. Because of the low level of serial communication, we only have three module layers represented here. We can think of the application—the `TodoX` example, for instance—as the protocol module. Applications at this level communicate with the device through these three layers (recall them from Chapter 4):

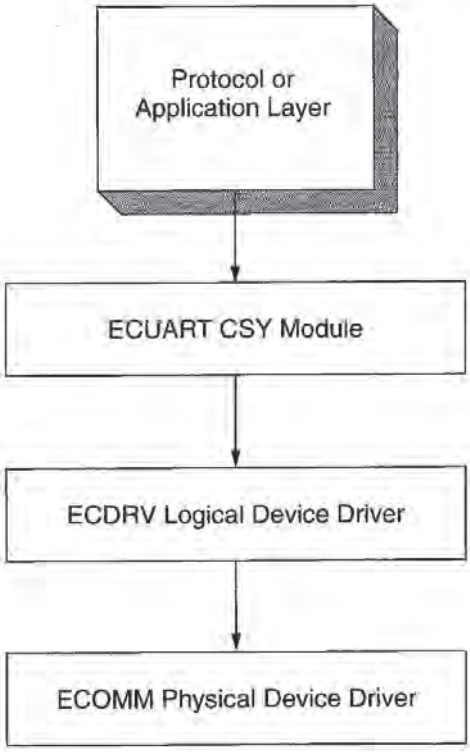

Figure 9.1 Module/driver structure for serial communication.

- The CSY module handles low-level, device independent software protocol issues. For example, data framing and buffering are issues handled here.
- The LDD module handles device-dependent hardware protocol issues, a hardware level counterpart to the CSY module. Any issues that have to do with data packets or formats are implemented again here for hardware.
- The PDD module is concerned only with moving bits from the LDD module to the hardware device. All issues other than managing the bit stream are ignored.

So if we want to change the device we use for communication, while maintaining the same method of communication, we only have to make some small changes to this structure. First, if the method of communication is the same—serial communication in each case—then the protocol module or application remains the same. We might change buffering or data frame formats or error control functions, so the CSY module will likely change. If our hardware is designed correctly—with the same interfaces—then the LDD and PDD modules can remain the same. We will use different device drivers when we open a different device.

It is this modular design that makes Symbian OS easy to extend and program (once we learn the basics). It means that extensions to the operating system only require installing new modules and loading them at the correct times. It also means that programming interfaces will remain consistent and applications will change little, if at all.

9.2 Serial Communication via the IR Port

Treating the IR port as a serial port is a great example of this operating system modularity. Writing an application that uses the IR port in a serial mode *is no different* in terms of the API than writing an application for the serial port. The patterns we developed in the last chapter apply directly; the API function calls work the same; the same concepts of asynchronous calls and active objects apply as well.

The major change from using the serial port is that different modules and drivers are to be used. Specifically, we need `IRCOMM.CSY` instead of `ECUART.CSY` and we need to open the `IRCOMM` device instead of `COMM::0`. At the risk of repetition, nothing else changes. The stack uses the same logical device driver and the same physical device driver.

Let's consider the `TodoX` example. We can subclass the `TodoXferIRSerial` class from `TodoXferSerialAO`, the serial example's active object class. The definition is below:

```
class TodoXferIRSerial : public TodoXferSerialAO
{
public:
   TodoXferIRSerial(CTodoXAppUi* aAppUi);
   ~TodoXferIRSerial();

public:
   static TodoXferIRSerial* NewLC(CTodoXAppUi* iAppUi);
   static TodoXferIRSerial* NewL(CTodoXAppUi* iAppUi);
```

```
public:
   TBool OpenL();
   TBool OpenL(TDesC&);
}
```

We need a class constructor (if only to call the superclass's constructor) and I have included the other functions only for completeness (so that one can use `OpenL()` with no arguments, for instance). We use all the other functions from the `TodoXferSerialAO` class, including `Read()`, `Write()`, `ReadItem()`, and `WriteItem()`.

We initialize the protocol differently as we load a different communication module to initialize the port, as this fragment from the `SendItem()` function of the `TodoX` example show:

```
void CTodoXAppUi::SendItem(TInt aIndex)
{
   iReader->ReadEntryL(iEntryIDs[aIndex]);

   switch (iXferMode) {

    ...
      case EIRSerialMode:
      case EIRSocketMode:
         iXferPort->Init(_L("IRCOMM"));
         iXferPort->OpenL();
         iXferPort->Configure();
         break;
    ...
}
```

So that you can see the similarity, I have included the `OpenL()` code below.

```
TBool TodoXferIRSerial::OpenL()
{
   return OpenL(_L("IRCOMM::0"));
}

TBool TodoXferIRSerial::OpenL(TDesC &aPortName)
{
   TInt result;

   result = commPort.Open(server,aPortName,
      ECommExclusive);
   User::LeaveIfError(result);

   return ETrue;
}
```

To convert our serial port `TodoX` example, that is all that is required. We have targeted the code at a new device and, since that device uses the same communication method as our serial example, this is all we have to do.

9.3 vCalendar Objects

In Chapter 8, I said I would deal with vCalendar objects in this chapter. However, this section is simply a teaser. The real meat of handling vCalendar objects appears in Chapter 13, where we discuss messaging. Here, I will overview some issues.

To send vCalendar objects over serial I/O, we need to handle them as text. Handling vCalendar objects in a textual form is a little difficult in Symbian OS. These objects are designed as BIO message content, not really as objects with textual content read by humans. Because of this, there are few function calls that will generate textual vCalendar representations.

One such function comes from the `CAgnModel` class, the class that lies at the center of Agenda item handling. Its definition is below:

```
void ExportVCalL(RWriteStream &aWriteStream,
    CAgnEntry* aEntry,
    const CVersitParser::TVersitCharSet aCharSet =
    CVersitParser::EUTF8CharSet);

void ExportVCalL(RWriteStream &aWriteStream,
    CArrayFixFlat<TAgnEntryId>* aEntryIdList,
    const CVersitParser::TVersitCharSet aCharSet =
    CVersitParser::EUTF8CharSet);
```

The `ExportVCalL()` function takes an entry or an array of entries and writes the textual representation to the `aWriteStream` object. Objects that derive from the `RWriteStream` class include both file and memory streams, so Agenda entries can be exported to files or memory as vCalendar objects.

Extend `TodoX` with Textual vCalendar Objects

One way to test how well you understand the concepts to this point is to extend the `TodoX` example to use textual vCalendar objects. You should call the (now empty) function call `SendItemVCal(CAgnEntry *aEntry)`, which should convert the vCalendar entry to text and use the established protocol to

send it. You also need to implement the `ReceiveItemVCal()` function that returns a pointer to a `CAgnEntry` object.

To do this, you will need to use the

```
WriteVCalObject(CAgnEntry* aEntry, RWriteStream aStream)
```

function from the `AgendaFileReader` class. This will convert the vCalendar object to text and write it to the `RWriteStream` object. If you examine the implementation of the `WriteVCalObject()` function, you will see it is implemented in terms of `CAgnModel::ExportVCalL()`.

9.4 Summary

This chapter detailed the techniques needed to use IR serial communications. The details, however, were not complicated; IR serial communications is based heavily on serial port communications, and very few changes need to be implemented. The chapter concluded with a brief discussion of the textual representation of vCalendar objects.

The next chapter is on sockets, a concept that is vital to programming the rest of the protocols in Symbian OS communications architecture. We revisit IR communications using sockets in Chapter 11.

10

Using Sockets

I enjoy the American television show *Star Trek*. Since its debut in 1967, it has gone through many changes, through five franchises, with a constant stream of new technology popping up in the show. Of all the innovations in the show, one of the most incredible inventions is the use of a "language translation matrix". Somehow, every person on the show speaks the same language—people from every country on earth and every alien species. While this is never explained on the show, it is implied that this is due to a computer (usually a starship's computer) using a translation matrix to translate one language to another in real time. It is amazing to me that each person hears only this translated speech—not the original source—and with no apparent artificial implants! I speak English and you hear Klingon! That's amazing technology!

Sockets are a bit like this language translation matrix. On one side, we might have an application that simply wants to send data—correctly, efficiently, rapidly—over an IR port. On the other side, another application wants to receive this data as it was sent—text, bytes, whatever. In between the two endpoints, there is much transformation that goes on—using a protocol stack as a translation matrix. Sockets allow the applications to ignore the translation matrix/protocol stack.

We will overview sockets in this chapter. Since sockets feature prominently in most of the communication methods for the rest of this book, a close look is warranted. We will first introduce the idea behind sockets, then examine three central implementations: the socket server, socket objects themselves, and name/address resolution.

10.1 Introduction to Sockets

Serial communication, good as it is, is still the raw exchange of data between two points. It can be considered, at best, a data link layer protocol. It is an effective—but limited—means of communication that allows data transfer over a physical medium. Its limitations are many, including a restricted transfer rate and very little error detection. To implement services like flow control and error correction, and to take advantage of more sophisticated communication techniques like multipoint communication, we must gain access to higher level communication protocols and use hardware devices through those protocols. We use sockets to gain this protocol access.

Sockets were invented by the designers of Berkeley Unix and were first used as a way to access TCP/IP protocols. In the Berkeley terminology, a socket is an "endpoint for communication". By itself, as an endpoint, a socket is not very useful. But when connected to another socket on another computer, the pair becomes a communication channel that uses a protocol to transfer data. You can think of sockets as two ends of a conversation and the protocol as the translator.

In fact, to write code that uses upper layer protocols—say TinyTP for IrDA—you can no longer deal with the device directly. This would be counterproductive; it would involve implementing protocol concepts that have already been implemented. Sockets allow us to take advantage of the protocol implementations already built for us by Symbian.

Sockets depend on four things: protocol modules, a socket server, a socket object, and a name resolution service.

10.2 Protocol Modules

To implement each of the many protocols it uses, Symbian OS uses protocol modules. We introduced protocol modules back in Chapter 3. You can think of them in the same way you do device drivers: there is a common programming interface that is implemented differently by the various protocol modules. In fact the analogy to device drivers works further: protocol modules are loaded and unloaded; they can be supplied by third party vendors implementing new protocols; and they are used by a server as loadable libraries to implement protocol stacks. For example, the various protocols in the TCP/IP suite are implemented by the `TCP.PRT` module and Bluetooth protocols are implemented by the `BT.PRT` module.

Loading a protocol module is usually automatic. The first time a socket that uses a particular protocol is opened, that protocol module is loaded. You can also load protocol modules programmatically through an `RSocketServ` object. Unloading a protocol module is also automatic, usually at the discretion of the operating system. However, an API exists that can be used to unload a protocol module as well.

Look for Yourself

Symbian OS uses a configuration file to specify which protocol modules are available and what they are used for. When it is needed—it is only used for additional third party protocol modules and not for built-in ones—it lists the protocol and the modules used for it.

Look on your emulator or on your hardware Symbian OS device. Look it up as `C:\System\data\esock.ini`. I have included a snippet of such a file from a Psion netBook:

```
[sockman]
protocols= plp,irmux,tinytp,tcp,udp,icmp,ip

[tcp]
filename= TCPIP.PRT
index= 4
bindto= ip,icmp

[udp]
filename= TCPIP.PRT
index= 3
bindto= ip,icmp

...
[nifman]
default= netdial

[agents]
ethernet= ethagt.agt
netdial= netdial.agt

[interfaces]
ethernet= ethint.nif
irlan= ethint.nif

[drivers]
ethernet= ethcard.drv
irlan= irlan.drv
```

10.3 RSocketServ: The Socket Server

Just as applications use a communications server to talk through CSY modules to hardware devices, they must also use a socket server to implement access through protocol modules to use protocols. The RSocketServ class is the class that implements the socket server.

In order to create a session with the socket server, an application must first connect to it. Connections are made by the following RSocketServ class function

```
TInt Connect(TUint aMessageSlots =
   KESockDefaultMessageSlots);
```

Calling the Connect() function will establish a connection with the socket server (making sure that it is running) and allocate a number of message slots for that connection. A message slot is used by a read or write operation. Since a socket can perform a read and a write asynchronously, a socket will usually require two message slots. The number of sockets you will use through the socket server will determine the number of message slots in the connection. The default number (if one is not given in the Connect() call) is eight.

While it is not necessary, protocol modules can be loaded and unloaded programmatically. This may be desirable if rapid response to all socket operations is extremely important. Without preloading, a socket server will load a protocol module with the first socket creation that uses the protocol that module implements. By loading modules at the start of an application, this load time can be redistributed to the beginning of the application. The function below loads a protocol module:

```
void StartProtocol(TUint anAddrFamily, TUint aSockType,
   TUint aProtocol,TRequestStatus &aStatus);
```

A call to StartProtocol() must give four parameters: the *family* of protocols that the caller wants (such as KAfInet for Internet protocols or KAfPlp for PLP protocols), the type of socket that will be used (such as a stream-based or datagram-based socket), the specific protocol selected from the protocol family (e.g, KProtocolInetICMP or KProtocolPlpLink), and a status variable we can use to monitor the operation (it is an asynchronous operation).

A socket server can be forced to unload a protocol module by the following function:

```
void StopProtocol(TUint anAddrFamily,TUint aSockType,
   TUint aProtocol,TRequestStatus &aStatus);
```

The parameters have the same meaning as with `StartProtocol()`.

Once we are finished with a socket server, we close the connection to it via its `Close()` function:

```
void Close();
```

10.4 RSocket: Using Sockets

Sockets are implemented via the `RSocket` class. Sockets have certain properties that must be chosen as they are created and have certain patterns to their use. They work differently if a server is using them than if a client is using them.

10.4.1 Socket Properties

Sockets have certain properties, chosen based on how they are used. A socket is either *connected* or *connectionless*. A connected socket maintains a *virtual connection* between the two endpoints. This means that address information for the remote endpoint needs to be given only once, and that each access to the connection can be done without respecifying this information. A connectionless socket forces the application to specify the remote endpoint information each time it is used and has no virtual connection. A connected socket is easier to use and is more reliable yet requires higher overhead in its implementation. Connected sockets use mechanisms to ensure that data arrives at the remote endpoint in the exact order they were sent, and that they arrive error-free or do not arrive at all. Connectionless sockets make no such guarantees about data arrival or reliability.

Connected sockets are implemented with *streams*. A stream is a logical connection between two endpoints that implements the properties of a connected socket. Specifically, streams implement the following properties:

- *Reliability:* With a stream, data are delivered accurately (as they were sent) without error or are not delivered at all. If there is no data delivery, this is detected, and the socket owner is notified.
- *Error control:* Errors are detected automatically, and the remote endpoint is usually asked to retransmit the data packet that had the error. Maintaining error control usually involves using checksums on data packets and forcing remote endpoints to acknowledge when they receive packets.

- *Ordered delivery:* As we have discussed before, data that flows between two endpoints is typically broken up and sent as fragments. Streams make sure those fragments arrive at their destination in the order they were sent and that the larger data packets are reassembled correctly.

Connectionless sockets are implemented with *datagrams*. Recall that a datagram is another name for a single data packet. A connectionless socket sends individual datagrams between endpoints rather than using a logical connection. No acknowledgement is required; no extensive error control or ordering mechanisms are used.

The reliability of connected sockets comes at a price. There are more protocol layers involved and hence more protocol overhead. There is more communication between endpoints and hence more data traffic. Connected sockets are implemented with streams; they are more complex than connectionless sockets.

10.4.2 Patterns in Socket Usage

There are patterns in how applications use sockets. In fact, these emulate the patterns we discussed back in Chapter 8 on how to use the serial port.

Initialization

With sockets, initializing "the system" means conversing with the socket server. We must start the server (if we need to) and establish a connection to it. To do this, we need to create an `RSocketServ` object and call its `Connect()` function:

```
RSocketServ socksvr;

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

If the return code is anything other than `KErrNone`, something has gone wrong, and we must handle the error. Once the current active object has connected to the socket server, that server can be used in further calls that need to obtain sockets.

Opening the socket

Instead of opening a communication port, as we did in Chapters 8 and 9, we now must open (and converse with) a socket. Sockets are designed to have an API that closely reflects the API

of a communications port, so the following functions should look familiar:

```
TInt Open(RSocketServ &aServer,TUint addrFamily,
   TUint sockType,TUint protocol);
TInt Open(RSocketServ &aServer,const TDesC &aName);
TInt Open(RSocketServ &aServer);
```

Each of these calls opens a socket for use by an application. The first form is the most specific and allows you to completely specify a socket. There are four parameters: a reference to a connected socket server, the address family of the socket (we discussed this previously for protocol loading in socket servers; an example is `KAfInet` for Internet protocols), the type of socket that will be used (such as a stream-based or datagram-based socket), and the specific protocol selected from the protocol family (e.g, `KProtocolInet-ICMP` or `KprotocolPlpLink`). The socket type is chosen from the following choices (from `es_sock.h`):

```
const TUint KSockStream=1;
const TUint KSockDatagram=2;
const TUint KSockSeqPacket=3;
const TUint KSockRaw=4;
```

The first two implement stream-based and datagram-based sockets, respectively. The third choice represents a middle ground between streams and datagrams, enforcing sequential packet delivery without all the other trappings of connected sockets. Finally, the last choice allows a socket connection with just a raw datastream—a situation where you would "build your own" packets. We will demonstrate this type of socket in Chapter 12.

The second form of the `Open()` function opens a socket and configures it by the name of the protocol given in the second parameter. This is the name of the protocol as given to it by Symbian OS and comes from the configuration file `esock.ini` in the emulator.

The third form of the `Open()` function creates a blank socket for use by application servers. The actual configuration for the socket is set when a connection is received from a client. This is covered in the next section.

Configuring and connecting sockets

Once a socket is open, it still needs to be connected to another "communication endpoint"—another socket. This is handled differently depending on the type of socket: connectionless/datagram-based or connected/stream-based.

Configuring connectionless sockets Connectionless sockets have a very brief, one-time connection with the remote socket. We configure connectionless sockets by giving them a *local* address and using special versions of I/O calls that include the remote socket's address. This is the minimal configuration that we can have.

We assign a local address to a socket through the following function from the `RSocket` class:

```
TInt Bind(TSockAddr &anAddr);
```

A call to `Bind()` will give the socket local information about the address and protocol it is supposed to represent, derived from the `anAddr` parameter. This information is built into the `TSockAddr` class and will be different for different protocols (addresses for TCP/IP, for example, are different than for Bluetooth).

Once a connectionless socket is bound to a local address, it is ready to send or receive data. These sockets can now be used with calls that specify the remote address, like `SendTo()` or `RecvFrom()`.

Configuring connected sockets Connected sockets are configured with the address of the remote socket, so that they remain tied together for the duration of the connection. The way this is done will be different based on whether the socket will be used for a server or a client application.

Client sockets are the easiest to use. A client socket makes a call to one of the functions below:

```
void Connect(TSockAddr &anAddr,TRequestStatus &aStatus);
void Connect(TSockAddr &anAddr,
   const TDesC8 &aConnectDataOut,TDes8 &aConnectDataIn,
   TRequestStatus &aStatus);
```

Both calls require an address to connect the socket to and a status variable to monitor the progress of the connection. Connecting is an asynchronous operation, and there may be some waiting involved for the remote side to make its connection. The effect of both calls, upon return, is to connect the socket to a server and to create a channel, accessible through the socket. The second form of the `Connect()` function sends (through `aConnectDataOut`) and receives (through `aConnectDataIn`) some initial data upon connection.

Servers must go through some additional steps. A server must listen for a connection from a client, accept the connection, and then "shape" a socket to fit the incoming connection so that

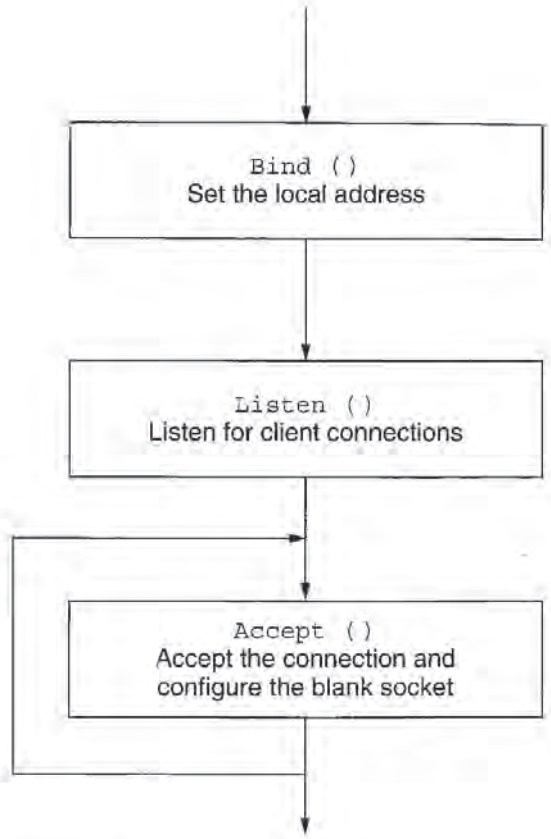

Figure 10.1 Configuration of an application server socket.

communication is possible. A server uses blank sockets, crafting them to fit the client's needs.

A server's sequence is depicted in Figure 10.1. A blank socket is bound to a local address (using the `RSocket::Bind()` function). Then the application must listen for incoming connections that specify the address bound to this socket. Once an incoming connection has been detected, the application must accept the incoming connection and configure a blank socket to service that connection. Once a connection is accepted, the socket channel is created, and data may flow between the two endpoints.

Listening is done through the `Listen()` function:

```
TInt Listen(TUint qSize);
TInt Listen(TUint qSize,const TDesC8 &aConnectData);
```

Listening for multiple connections is handled concurrently from a queue of size `qSize`. This queue will contain `qSize` number of open connections waiting to be connected to sockets. Connection requests from remote devices that occur when the queue is full are rejected. The second form will receive some data through

the connection when one is made. Notice that this call simply configures the queue for collecting incoming requests. It is not asynchronous and returns immediately with a result code.

Accepting an incoming connection joins a blank socket to the connection request through the following functions:

```
void Accept(RSocket &aBlankSocket,
   TRequestStatus &aStatus);
void Accept(RSocket &aBlankSocket,TDes8 &aConnectData,
   TRequestStatus &aStatus);
```

This call will block until an incoming connection is detected. When a connection is detected, the blank socket specified by `aBlankSocket` will be connected and configured with addressing information. After this call completes, the newly configured socket can be used to converse with the remote device.

An example of an application server's use of sockets is in the code fragment below:

```
RSocketServ iSockSvr;
RSocket iListener, iServiceSocket;
TInetAddr iAddress;
TRequestStatus iStat;

//... code to create sockets and connect to the server

// Open the listener socket
err = iListener.Open(iSockSvr, KAfInet, KSockStream,
   KundefinedProtocol);
User::LeaveIfError(err);

// Open the blank socket
err = iServiceSocket.Open(iSockSvr);
User::LeaveIfError(err);

// Bind the listener socket to ANY address from port 3100
iAddress = new TInetAddr(KInetAddrAny, 3100);
iListener.Bind(iAddress);

// Set up a queue of 5 possible connections at a time
iListener.Listen(5);
// And wait for a connection, joining the blank socket to
// to the incoming request
iListener.Accept(iServiceSocket, iStat);
SetActive();
```

Note that we are assuming an active object here, hence the use of `SetActive()`. In this example, the code waits for devices using TCP/IP through port 3100.

It bears repeating that two sockets are involved in an application server. One socket is the listener socket and accepts incoming connection requests. The second socket is a blank socket that is joined to the incoming request once it has been accepted. It is through this second socket that active data exchange takes place.

Finally, note that the `Accept()` function implements an asynchronous operation and that it can be cancelled with the following function:

```
void CancelAccept();
```

Exchanging data over sockets

Reading and writing data using sockets is designed to closely resemble what we did with direct communication with hardware ports. The API is very similar but adds a few convenience functions.

Connectionless sockets Reading and writing with datagrams means that the address (host and port) of the remote side of the connection must be specified with the I/O request. This takes special functions that are based on `Read()` and `Write()` but also use address information.

Reading from a connectionless socket uses the `RecvFrom()` function:

```
void RecvFrom(TDes8 &aDesc,TSockAddr &anAddr,TUint flags,
   TRequestStatus &aStatus);
void RecvFrom(TDes8 &aDesc,TSockAddr &anAddr,TUint flags,
   TRequestStatus &aStatus,TSockXfrLength &aLen);
```

Both calls give a buffer, an address of the device to receive the data from, some flags that configure the read, and a status variable to monitor the receipt of data. As usual, the first form will try to fill its buffer, and the second form will only read `aLen` bytes. The behavior of this function is to block until a datagram from the sender specified in the `anAddr` parameter is received.

Writing to a connectionless socket is done through the `SendTo()` function:

```
void SendTo(const TDesC8 &aDesc,TSockAddr &anAddr,
   TUint flags,TRequestStatus &aStatus);
void SendTo(const TDesC8 &aDesc,TSockAddr &anAddr,
   TUint flags,TRequestStatus &aStatus,
   TSockXfrLength &aLen);
```

Again, we have a buffer containing the data to send, an address to send the datagram to, some flags to control the transfer, and a status

variable for monitoring. The first form sends the entire buffer; the second form only sends `aLen` bytes. The blocking behavior here is controlled by the flags in the call; the default behavior is to send without waiting for receipt.

Connected sockets Because we do not need to include address information with connected socket I/O, we have a bit more flexibility in the functions we can use.

At the lowest level, `Read()` and `Write()` functions work as we have defined them before. However, they have parameters in a different order:

```
void Read(TDes8 &aDesc,TRequestStatus &aStatus);
void CancelRead();
void Write(const TDesC8 &aDesc,TRequestStatus &aStatus);
void CancelWrite();
```

Notice that each comes with a canceling function, because each is asynchronous. In addition, notice that each comes in only one version—lacking a version with the buffer length specification.

The more flexible way to do socket I/O is to use `Recv()` and `Send()` functions. These functions include buffer length specifiers and flags to control the I/O, along with the other required parameters:

```
void Recv(TDes8 &aDesc,TUint flags,
   TRequestStatus &aStatus);
void Recv(TDes8 &aDesc,TUint flags,
   TRequestStatus &aStatus,TSockXfrLength &aLen);
void CancelRecv()
void Send(const TDesC8 &aDesc,TUint someFlags,
   TRequestStatus &aStatus);
void Send(const TDesC8 &aDesc,TUint someFlags,
   TRequestStatus &aStatus,TSockXfrLength &aLen);
void CancelSend();
```

Each function really looks like (and is implemented by) lower level `Read()` and `Write()` functions. Each function is complemented by its own canceling function.

There is one more function that can be used to receive data on a connected port. As with reading from a hardware port, we can also be flexible with the amount of data we receive and process. The function below allows us to do this:

```
void RecvOneOrMore(TDes8 &aDesc,TUint flags,
   TRequestStatus &aStatus,TSockXfrLength &aLen);
```

This call will cause a read of as much data as is available in the input buffer. The number of bytes read is returned in the `aLen` variable.

Closing sockets

Sockets are terminated using a few more steps than raw devices. This is so that the protocol layers involved in the socket connection may also shutdown.

First, the socket may have pending connection requests or buffers with data ready to be retrieved. To close a socket gracefully, we should use the `CancelAll()` function:

```
void CancelAll();
```

This will cancel any asynchronous operations that may be waiting for data or connections.

The next step is to close the socket. There are two ways to do this, depending on whether synchronous or asynchronous operations are desired. The synchronous way to do this is to use `Close()`:

```
void Close();
```

This will terminate the socket and cause the operating system to release any resources dedicated to it. The asynchronous way to do it is to use the `Shutdown()` function:

```
void Shutdown(TShutdown aHow,TRequestStatus &aStatus);
void Shutdown(TShutdown aHow,const TDesC8
   &aDisconnectDataOut,TDes8 &aDisconnectDataIn,
   TRequestStatus &aStatus);
```

The first form informs the system how to shut the socket down and gives a status variable for monitoring. The "how" of the shutdown process is given by the definition of `TShutdown`:

```
enum TShutdown {ENormal,EStopInput,EStopOutput,EImmediate}
```

The "normal" way is to drain the socket, waiting for both input and output buffers to empty. The other functions will stop the input but drain the output, drain the input but stop the output, or just stop both. The second form of `Shutdown()` allows disconnect messages to be sent and received.

The typical way to close a socket is to use the synchronous `Close()` function. This works best when there are no pending

operations or buffered data waiting (using `Close()` in these situations will stop the operations and lose the data). The asynchronous functions should be used when some waiting for operations or buffers is necessary.

10.4.3 Some Odds and Ends

There are a few socket maintenance functions and some convenience functions that bear mention:

- Socket options are configuration parameters that control the behavior of sockets. There are generic options that apply to all sockets (such as a debug mode) and there are specific options that apply to specific protocols (such as raw mode for TCP/IP sockets). There are four functions that work with options:

```
TInt GetOpt(TUint anOptionName,TUint anOptionLevel,
   TDes8 &anOption);
TInt GetOpt(TUint anOptionName,TUint anOptionLevel,
   TInt &anOption);
TInt SetOpt(TUint anOptionName,TUint anOptionLevel,
   TInt &anOption);
TInt SetOpt(TUint anOptionName, TUint anOptionLevel,
   const TDesC8 &anOption=TPtrC8(NULL,0));
```

  These functions will get an option value—in either string or integer form (hence the two versions)—or will set an option. We can also set options asynchronously. The `Ioctl()` function does this for us, along with its canceling function:

```
void Ioctl(TUint aCommand,TRequestStatus &aStatus,
   TDes8* aDesc=NULL, TUint aLevel=KLevelUnspecified);
void CancelIoctl();
```

- The local address and port of a socket can be retrieved through the following function calls:

```
void LocalName(TSockAddr &anAddr);
TUint LocalPort();
```

  The hostname can be retrieved using the `RHostResolver` class, but the you must use the functions above for the address and port.

- You can retrieve the address of the remote endpoint of a socket using

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

- You can obtain information about the protocol a socket is using by calling the following function:

```
TInt Info(TProtocolDesc &aProtocol);
```

This call returns an instance of the `TProtocolDesc` class, listed below, which contains valuable information about a protocol:

```
struct TProtocolDesc
{
public:
  TProtocolName iName;
  TUint iAddrFamily;
  TUint iSockType;
  TUint iProtocol;
  TVersion iVersion;
  TByteOrder iByteOrder;
  TUint iServiceInfo;
  TUint iNamingServices;
  TUint iSecurity;
  TInt iMessageSize;
};
```

10.5 Resolving Host Names and Addresses

Computers that communicate with one another must be able to tell each other apart. They must know, for example, if data is coming for them or for some other device. For identification reasons, communication devices are given an *address*, some identifying data sequence that uniquely identifies a device for the protocol it is using. Addresses are protocol-dependent and can look convoluted. TCP/IP addresses, for example, are currently four sets of numbers, tied together with dots: e.g., `192.168.3.76`. To complicate matters, services on computers are typically identified by port number, adding to the complexity of addressing the SMTP port as `192.168.3.76:25`.

These can be hard for humans to remember, however. So names are given to devices. It is easier, for instance, to remember ***http://www.symbian.com*** than to remember `195.224.64.203`. Names are also given to services, allowing port 23 to be addressed as the Telnet port and port 80 as the HTTP port and so on.

Tying host and service names to their address counterparts is the job of the *resolver*. The resolver provides access to lookup services that can look up addresses when given names and vice versa. There are resolution services for many protocols; perhaps the most familiar is the Domain Name Service (DNS) for TCP/IP.

Symbian OS provides a name resolution interface through the `RHostResolver` class. Like the `RSocket` class, the `RHostResolver` class depends on a connection with a socket server. Once you have connected to a socket server, using the name resolution interface looks much like using a socket: you must open a connection to the host resolver, interact with it, and then close the connection.

The function call for opening a connection to the host resolution service should look familiar:

```
TInt Open(RSocketServ &aSocketServer,TUint anAddrFamily,
   TUint aProtocol);
```

You must specify a connected socket server, an address family and a protocol as parameters. The function returns a code that indicates success or failure. As an example, the fragment of code below opens a connection to the resolution service:

```
err = socksvr.Connect();
User::LeaveIfError(err);
err = resolver.Open(socksvr, KAfInet, KProtocolInetTcp);
User::LeaveIfError(err);
```

This fragment opens a connection to the name resolver for TCP/IP using the Internet address family.

With an open resolver, we can make queries through several functions:

```
void GetByName(const TDesC &aName,
   TNameEntry &aResult,TRequestStatus &aStatus);
TInt GetByName(const TDesC &aName,TNameEntry &aResult);
void GetByAddress(const TSockAddr &anAddr,
   TNameEntry &aResult,TRequestStatus &aStatus);
TInt GetByAddress(const TSockAddr &anAddr,
   TNameEntry &aResult);
```

Each of these functions comes in synchronous and asynchronous versions; a status variable is included in the asynchronous versions.

`GetByName()` takes a name and returns addressing information by way of a `TNameEntry` structure, which itself is an array of `TNameRecord` structures. `TNameRecord` looks like this:

```
class TNameRecord
{
public:
  inline TNameRecord();
  enum {EAlias =0x00000001};
  THostName iName;
```

```
  TSockAddr iAddr;
  TInt iFlags;
};
```

Basically, this is a hostname/address pair. It is put into an array because there may be several pairs associated with a single device.

`GetByAddress()` goes the opposite direction from `GetByName()`: it takes an address and returns the name of the device in the given `TNameEntry` class.

If there are multiple names or addresses for a device, the `Next()` function will bring up the next value:

```
void Next(TNameEntry &aResult,TRequestStatus &aStatus);
TInt Next(TNameEntry &aResult);
```

Sometimes a device will support multiple addresses for an interface, have multiple interfaces, or simply have aliases by which it is known. `Next()` must return a status code of `KErrNone` for the next name/address pair to be valid.

Multiple Hostnames and DNS

In the TCP/IP network area, it is common for a computer to have multiple hostnames with one IP address. Some examples are

- A computer that has one name to computers on an internal network and another name to those on an external network—like a firewall.
- A Web server that has a name to match local computers (e.g., ***http://frodo.middleearth.biz***) and a service name to match services it offers (e.g. ***http://www.middleearth.biz*** or ***http://mail.middleearth.biz***)

Find computers on your local network with multiple names.

There is another use for name resolution that goes beyond the standard use for resolving names and addresses. Device discovery in IrDA communications uses host resolution functions, as we will see in the next chapter.

```
void GetHostName(TDes &aName,TRequestStatus &aStatus)
TInt GetHostName(TDes &aName)
```

Finally, `GetHostName()` focuses on the local device. It will return the name of the local host if there is a name for it.

10.6 Summary

Sockets are the means by which we take advantage of protocols designed for higher level communications access. In this chapter, we overviewed sockets and protocol modules, then focused on three elements used in the socket system:

- The *socket server*, which handles the mechanics of allocating socket resources and loading protocol modules
- The *socket* itself and all the ways of using it provided by Symbian OS
- The *host resolver*, which handles the mapping between names and addresses for protocols

The next chapter takes a new look at IrDA communications by accessing different protocols through sockets. IrDA provides a great example that illustrates the ideas we discussed here.